

# **Cadence<sup>®</sup> Verilog<sup>®</sup> -A Language Reference**

**Product Version 13.1.1**  
**April 2014**

© 1996–2014 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

MMSIM contains technology licensed from, and copyrighted by: C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh © 1979, J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson © 1988, J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling © 1990; University of Tennessee, Knoxville, TN and Oak Ridge National Laboratory, Oak Ridge, TN © 1992-1996; Brian Paul © 1999-2003; M. G. Johnson, Brisbane, Queensland, Australia © 1994; Kenneth S. Kundert and the University of California, 1111 Franklin St., Oakland, CA 94607-5200 © 1985-1988; Hewlett-Packard Company, 3000 Hanover Street, Palo Alto, CA 94304-1185 USA © 1994, Silicon Graphics Computer Systems, Inc., 1140 E. Arques Ave., Sunnyvale, CA 94085 © 1996-1997, Moscow Center for SPARC Technology, Moscow, Russia © 1997; Regents of the University of California, 1111 Franklin St., Oakland, CA 94607-5200 © 1990-1994, Sun Microsystems, Inc., 4150 Network Circle Santa Clara, CA 95054 USA © 1994-2000, Scriptics Corporation, and other parties © 1998-1999; Aladdin Enterprises, 35 Eyal St., Kiryat Arye, Petach Tikva, Israel 49511 © 1999 and Jean-loup Gailly and Mark Adler © 1995-2005; RSA Security, Inc., 174 Middlesex Turnpike Bedford, MA 01730 © 2005.

All rights reserved. Associated third party license terms may be found at *install\_dir/doc/OpenSource/\**

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

**Trademarks:** Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

**Restricted Permission:** This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

**Disclaimer:** Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

**Restricted Rights:** Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor

---

# Contents

---

<u>Preface</u> .....	21
<u>Related Documents</u> .....	21
<u>Typographic and Syntax Conventions</u> .....	22
<b>1</b>	
<b><u>Modeling Concepts</u></b> .....	25
<u>Verilog-A Language Overview</u> .....	26
<u>Describing a System</u> .....	27
<u>Analog Systems</u> .....	28
<u>Nodes</u> .....	28
<u>Conservative Systems</u> .....	29
<u>Signal-Flow Systems</u> .....	29
<u>Mixed Conservative and Signal-Flow Systems</u> .....	29
<u>Simulator Flow</u> .....	30
<b>2</b>	
<b><u>Creating Modules</u></b> .....	33
<u>Overview</u> .....	34
<u>Declaring Modules</u> .....	34
<u>Declaring the Module Interface</u> .....	36
<u>Module Name</u> .....	36
<u>Ports</u> .....	36
<u>Parameters</u> .....	39
<u>Defining Module Analog Behavior</u> .....	39
<u>Defining Analog Behavior with Control Flow</u> .....	41
<u>Using Integration and Differentiation with Analog Signals</u> .....	42
<u>Using Internal Nodes in Modules</u> .....	43
<u>Using Internal Nodes in Behavioral Definitions</u> .....	44
<u>Using Internal Nodes in Higher Order Systems</u> .....	44
<u>Instantiating Modules with Netlists</u> .....	45

## 3

<b><u>Lexical Conventions</u></b> .....	47
<u>White Space</u> .....	48
<u>Comments</u> .....	48
<u>Identifiers</u> .....	48
<u>Ordinary Identifiers</u> .....	49
<u>Escaped Names</u> .....	49
<u>Scope Rules</u> .....	49
<u>Numbers</u> .....	50
<u>Integer Numbers</u> .....	50
<u>Real Numbers</u> .....	52

## 4

<b><u>Data Types and Objects</u></b> .....	55
<u>Output Variables</u> .....	56
<u>Integer Numbers</u> .....	56
<u>Real Numbers</u> .....	57
<u>Converting Real Numbers to Integer Numbers</u> .....	57
<u>Strings</u> .....	58
<u>Parameters and Local Parameters</u> .....	58
<u>Specifying a Parameter Type</u> .....	61
<u>Specifying Permissible Values</u> .....	61
<u>Specifying Parameter Arrays</u> .....	62
<u>String Parameters</u> .....	63
<u>Parameter Aliases</u> .....	64
<u>Paramsets</u> .....	64
<u>Paramset Output Variables</u> .....	66
<u>Genvars</u> .....	66
<u>Natures</u> .....	68
<u>Declaring a Base Nature</u> .....	69
<u>Disciplines</u> .....	71
<u>Binding Natures with Potential and Flow</u> .....	71
<u>Compatibility of Disciplines</u> .....	72
<u>Net Disciplines</u> .....	75

## Cadence Verilog-A Language Reference

---

<u>Named Branches</u> .....	77
<u>Implicit Branches</u> .....	78
<u>Output Variables</u> .....	79

### 5

<u>Statements for the Analog Block</u> .....	81
<u>Analog Initial Block</u> .....	82
<u>Assignment Statements</u> .....	82
<u>Procedural Assignment Statements in the Analog Block</u> .....	82
<u>Branch Contribution Statement</u> .....	83
<u>Indirect Branch Assignment Statement</u> .....	85
<u>Sequential Block Statement</u> .....	86
<u>Conditional Statement</u> .....	86
<u>Case Statement</u> .....	87
<u>Repeat Statement</u> .....	88
<u>While Statement</u> .....	89
<u>For Statement</u> .....	89
<u>Generate Statement</u> .....	90

### 6

<u>Operators for Analog Blocks</u> .....	93
<u>Overview of Operators</u> .....	94
<u>Unary Operators</u> .....	95
<u>Unary Reduction Operators</u> .....	95
<u>Binary Operators</u> .....	96
<u>Bitwise Operators</u> .....	99
<u>Ternary Operator</u> .....	100
<u>Operator Precedence</u> .....	101
<u>Expression Short-Circuiting</u> .....	101
<u>String Operators and Functions</u> .....	101
<u>String Operator Details</u> .....	103
<u>String Function Details</u> .....	104

## 7

<b><u>Built-In Mathematical Functions</u></b> .....	111
<u>Standard Mathematical Functions</u> .....	112
<u>Trigonometric and Hyperbolic Functions</u> .....	113
<u>Controlling How Math Domain Errors Are Handled</u> .....	113

## 8

<b><u>Detecting and Using Analog Events</u></b> .....	117
<u>Detecting and Using Events</u> .....	118
<u>Initial_step Event</u> .....	119
<u>Final_step Event</u> .....	119
<u>Cross Event</u> .....	120
<u>Above Event</u> .....	122
<u>Timer Event</u> .....	124

## 9

<b><u>Simulator Functions</u></b> .....	125
<u>Announcing Discontinuity</u> .....	127
<u>Bounding the Time Step</u> .....	129
<u>Finding When a Signal Is Zero</u> .....	130
<u>Querying the Simulation Environment</u> .....	131
<u>Obtaining the Current Simulation Time</u> .....	131
<u>Obtaining the Current Ambient Temperature</u> .....	132
<u>Obtaining the Thermal Voltage</u> .....	132
<u>Querying the scale, gmin, and iteration Simulation Parameters</u> .....	132
<u>Probing of values within a sibling instance during simulation</u> .....	133
<u>Relating a Specific Frequency to a Source Name for RF</u> .....	134
<u>Detecting Parameter Overrides</u> .....	135
<u>Detecting Port Binding</u> .....	136
<u>Obtaining and Setting Signal Values</u> .....	137
<u>Accessing Attributes</u> .....	140
<u>Analysis-Dependent Functions</u> .....	141
<u>Determining the Current Analysis Type</u> .....	141
<u>Implementing Small-Signal AC Sources</u> .....	143

## Cadence Verilog-A Language Reference

---

<u>Implementing Small-Signal Noise Sources</u>	144
<u>Generating Random Numbers</u>	146
<u>\$random</u>	146
<u>\$arandom</u>	147
<u>Generating Random Numbers in Specified Distributions</u>	149
<u>Uniform Distribution</u>	149
<u>Normal (Gaussian) Distribution</u>	150
<u>Exponential Distribution</u>	151
<u>Poisson Distribution</u>	152
<u>Chi-Square Distribution</u>	153
<u>Student's T Distribution</u>	153
<u>Erlang Distribution</u>	154
<u>Interpolating with Table Models</u>	156
<u>Table Model File Format</u>	158
<u>Example: Using the \$table_model Function</u>	159
<u>Example: Preparing Data in One-Dimensional Array Format</u>	160
<u>Analog Operators</u>	161
<u>Restrictions on Using Analog Operators</u>	161
<u>Limited Exponential Function</u>	161
<u>Time Derivative Operator</u>	162
<u>Time Integral Operator</u>	162
<u>Circular Integrator Operator</u>	164
<u>Derivative Operator</u>	166
<u>Delay Operator</u>	167
<u>Transition Filter</u>	168
<u>Slew Filter</u>	171
<u>Implementing Laplace Transform S-Domain Filters</u>	173
<u>Implementing Z-Transform Filters</u>	178
<u>Displaying Results</u>	183
<u>\$strobe</u>	184
<u>\$display</u>	187
<u>\$write</u>	187
<u>\$debug</u>	188
<u>Specifying Power Consumption</u>	188
<u>Indicating Non-linearities to the Simulator</u>	189
<u>Working with Files</u>	191

## Cadence Verilog-A Language Reference

---

<u>Opening a File</u> .....	191
<u>Reading from a File</u> .....	194
<u>Writing to a File</u> .....	195
<u>Closing a File</u> .....	197
<u>Writing to a Variable</u> .....	197
<u>\$write</u> .....	197
<u>\$format</u> .....	197
<u>Simulator Control Functions</u> .....	198
<u>\$finish</u> .....	198
<u>\$finish current analysis</u> .....	199
<u>\$stop</u> .....	199
<u>\$fatal</u> .....	200
<u>\$error</u> .....	201
<u>\$warning</u> .....	201
<u>\$info</u> .....	201
<u>Obtaining the Trial Number for Monte Carlo Analysis</u> .....	201
<u>\$cds_get_mc_trial_number()</u> .....	201
<u>User-Defined Functions</u> .....	202
<u>Declaring an Analog User-Defined Function</u> .....	203
<u>Returning a Value From an Analog User-Defined Function</u> .....	205
<u>Calling a User-Defined Analog Function</u> .....	207
<u>Calling functions implemented in C</u> .....	208
<u>Import declaration</u> .....	208
<u>Loading C function from a Dynamic Link Library</u> .....	209

## 10

<u>Instantiating Modules and Primitives</u> .....	213
<u>Instantiating Verilog-A Modules</u> .....	214
<u>Creating and Naming Instances</u> .....	214
<u>Mapping Instance Ports to Module Ports</u> .....	215
<u>Connecting the Ports of Module Instances</u> .....	219
<u>Port Connection Rules</u> .....	220
<u>Overriding Parameter Values in Instances</u> .....	220
<u>Overriding Parameter Values from the Module Instance Statement</u> .....	222
<u>Overriding Parameter Values by Using Paramsets</u> .....	223



# Cadence Verilog-A Language Reference

---

<u>Instantiating Analog Primitives</u> .....	225
<u>Instantiating Analog Primitives that Use Array Valued Parameters</u> .....	225
<u>Instantiating Modules that Use Unsupported Parameter Types</u> .....	226
<u>Using Inherited Ports</u> .....	226
<u>Using an Inherited m Factor (Multiplicity Factor)</u> .....	228
<u>Setting an m Factor Directly on a Verilog-A Module</u> .....	230
<u>Using the \$mfactor System Function</u> .....	232
<u>\$mfactor Double-Scaling</u> .....	233
<u>Using \$mfactor Together with the Standard m Factor</u> .....	234
<u>Using \$mfactor Together with an Inherited m Factor</u> .....	235

## 11

<u>Controlling the Compiler</u> .....	237
<u>Using Compiler Directives</u> .....	238
<u>Implementing Text Macros</u> .....	238
<u>`define Compiler Directive</u> .....	238
<u>`undef Compiler Directive</u> .....	240
<u>Compiling Code Conditionally</u> .....	240
<u>`ifdef Compiler Directive</u> .....	240
<u>`ifndef Compiler Directive</u> .....	241
<u>Including Files at Compilation Time</u> .....	242
<u>Setting Default Rise and Fall Times</u> .....	242
<u>Resetting Directives to Default Values</u> .....	243

## 12

<u>AHDL Linter Checks</u> .....	245
<u>About the AHDL Linter Feature</u> .....	246
<u>Using the AHDL Linter Feature in APS, XPS and UltraSim</u> .....	246
<u>Identifying AHDL Linter Messages</u> .....	248
<u>Static AHDL Linter Message</u> .....	248
<u>Dynamic AHDL Linter Message</u> .....	248
<u>Filtering AHDL Linter Messages</u> .....	249
<u>Using the ahdhelp Utility</u> .....	249

## 13

### Using Verilog-A in the Cadence Analog Design Environment

251

<u>Creating Cellviews Using the Cadence Analog Design Environment</u> .....	252
<u>Preparing a Library</u> .....	252
<u>Creating the Symbol View</u> .....	254
<u>Using Blocks</u> .....	255
<u>Creating a Verilog-A Cellview from a Symbol or Block</u> .....	256
<u>Descend Edit</u> .....	260
<u>Creating a Verilog-A Cellview</u> .....	260
<u>Creating a Symbol Cellview from an Analog HDL Cellview</u> .....	263
<u>Using Escaped Names in the Cadence Analog Design Environment</u> .....	264
<u>Defining Quantities</u> .....	264
<u>spectre/spectreVerilog Interface (Spectre Direct)</u> .....	265
<u>Using Multiple Cellviews for Instances</u> .....	266
<u>Creating Multiple Cellviews for a Component</u> .....	267
<u>Modifying the Parameters Specified in Modules</u> .....	268
<u>Switching the Cellview Bound with an Instance</u> .....	272
<u>Example Illustrating Cellview Switching</u> .....	275
<u>Multilevel Hierarchical Designs</u> .....	285
<u>Including Verilog-A through Model Setup</u> .....	286
<u>Netlisting Verilog-A Modules</u> .....	286
<u>Hierarchical Verilog-A Modules</u> .....	286
<u>Using a Hierarchy</u> .....	288
<u>Using Models with Verilog-A</u> .....	290
<u>Models in Modules</u> .....	290
<u>Saving Verilog-A Variables</u> .....	291
<u>Displaying the Waveforms of Variables</u> .....	291

## 14

### Verilog-A Modeling Examples .....

<u>Electrical Modeling</u> .....	296
<u>Three-Phase, Half-Wave Rectifier</u> .....	296
<u>Thin-Film Transistor Model</u> .....	301

## Cadence Verilog-A Language Reference

---

<u>Mechanical Modeling</u> .....	307
<u>Car on a Bumpy Road</u> .....	308
<u>Gearbox</u> .....	315
<u>Computing a Moving or Sliding-Window Average</u> .....	321

### 15

<u>Nodal Analysis</u> .....	323
<u>Kirchhoff's Laws</u> .....	324
<u>Simulating a System</u> .....	325
<u>Transient Analysis</u> .....	325
<u>Convergence</u> .....	325

### 16

<u>Analog Probes and Sources</u> .....	327
<u>Overview of Probes and Sources</u> .....	328
<u>Probes</u> .....	328
<u>Port Branches</u> .....	328
<u>Sources</u> .....	329
<u>Unassigned Sources</u> .....	331
<u>Switch Branches</u> .....	331
<u>Examples of Sources and Probes</u> .....	334
<u>Linear Conductor</u> .....	334
<u>Linear Resistor</u> .....	335
<u>RLC Circuit</u> .....	335
<u>Simple Implicit Diode</u> .....	335

### 17

<u>Sample Model Library</u> .....	337
<u>Analog Components</u> .....	339
<u>Analog Multiplexer</u> .....	339
<u>Current Deadband Amplifier</u> .....	340
<u>Hard Current Clamp</u> .....	341
<u>Hard Voltage Clamp</u> .....	342
<u>Open Circuit Fault</u> .....	343

## Cadence Verilog-A Language Reference

---

<u>Operational Amplifier</u> .....	344
<u>Constant Power Sink</u> .....	345
<u>Short Circuit Fault</u> .....	346
<u>Soft Current Clamp</u> .....	347
<u>Soft Voltage Clamp</u> .....	348
<u>Self-Tuning Resistor</u> .....	349
<u>Untrimmed Capacitor</u> .....	351
<u>Untrimmed Inductor</u> .....	352
<u>Untrimmed Resistor</u> .....	353
<u>Voltage Deadband Amplifier</u> .....	354
<u>Voltage-Controlled Variable-Gain Amplifier</u> .....	355
<u>Basic Components</u> .....	356
<u>Resistor</u> .....	356
<u>Capacitor</u> .....	357
<u>Inductor</u> .....	358
<u>Voltage-Controlled Voltage Source</u> .....	359
<u>Current-Controlled Voltage Source</u> .....	360
<u>Voltage-Controlled Current Source</u> .....	361
<u>Current-Controlled Current Source</u> .....	362
<u>Switch</u> .....	363
<u>Control Components</u> .....	364
<u>Error Calculation Block</u> .....	364
<u>Lag Compensator</u> .....	365
<u>Lead Compensator</u> .....	366
<u>Lead-Lag Compensator</u> .....	367
<u>Proportional Controller</u> .....	368
<u>Proportional Derivative Controller</u> .....	369
<u>Proportional Integral Controller</u> .....	370
<u>Proportional Integral Derivative Controller</u> .....	371
<u>Logic Components</u> .....	372
<u>AND Gate</u> .....	372
<u>NAND Gate</u> .....	373
<u>OR Gate</u> .....	374
<u>NOT Gate</u> .....	375
<u>NOR Gate</u> .....	376
<u>XOR Gate</u> .....	377

## Cadence Verilog-A Language Reference

---

<u>XNOR Gate</u>	378
<u>D-Type Flip-Flop</u>	379
<u>Clocked JK Flip-Flop</u>	380
<u>JK-Type Flip-Flop</u>	382
<u>Level Shifter</u>	383
<u>RS-Type Flip-Flop</u>	384
<u>Trigger-Type (Toggle-Type) Flip-Flop</u>	385
<u>Half Adder</u>	386
<u>Full Adder</u>	387
<u>Half Subtractor</u>	388
<u>Full Subtractor</u>	389
<u>Parallel Register, 8-Bit</u>	390
<u>Serial Register, 8-Bit</u>	391
<u>Electromagnetic Components</u>	392
<u>DC Motor</u>	392
<u>Electromagnetic Relay</u>	393
<u>Three-Phase Motor</u>	394
<u>Functional Blocks</u>	395
<u>Amplifier</u>	395
<u>Comparator</u>	396
<u>Controlled Integrator</u>	397
<u>Deadband</u>	398
<u>Deadband Differential Amplifier</u>	399
<u>Differential Amplifier (Opamp)</u>	400
<u>Differential Signal Driver</u>	401
<u>Differentiator</u>	402
<u>Flow-to-Value Converter</u>	403
<u>Rectangular Hysteresis</u>	404
<u>Integrator</u>	405
<u>Level Shifter</u>	406
<u>Limiting Differential Amplifier</u>	407
<u>Logarithmic Amplifier</u>	408
<u>Multiplexer</u>	409
<u>Quantizer</u>	410
<u>Repeater</u>	411
<u>Saturating Integrator</u>	412

## Cadence Verilog-A Language Reference

---

<u>Swept Sinusoidal Source</u> .....	413
<u>Three-Phase Source</u> .....	414
<u>Value-to-Flow Converter</u> .....	415
<u>Variable Frequency Sinusoidal Source</u> .....	416
<u>Variable-Gain Differential Amplifier</u> .....	417
<u>Magnetic Components</u> .....	418
<u>Magnetic Core</u> .....	418
<u>Magnetic Gap</u> .....	419
<u>Magnetic Winding</u> .....	420
<u>Two-Phase Transformer</u> .....	421
<u>Mathematical Components</u> .....	422
<u>Absolute Value</u> .....	422
<u>Adder</u> .....	423
<u>Adder, 4 Numbers</u> .....	424
<u>Cube</u> .....	425
<u>Cubic Root</u> .....	426
<u>Divider</u> .....	427
<u>Exponential Function</u> .....	428
<u>Multiplier</u> .....	429
<u>Natural Log Function</u> .....	430
<u>Polynomial</u> .....	431
<u>Power Function</u> .....	432
<u>Reciprocal</u> .....	433
<u>Signed Number</u> .....	434
<u>Square</u> .....	435
<u>Square Root</u> .....	436
<u>Subtractor</u> .....	437
<u>Subtractor, 4 Numbers</u> .....	438
<u>Measure Components</u> .....	439
<u>ADC, 8-Bit Differential Nonlinearity Measurement</u> .....	439
<u>ADC, 8-Bit Integral Nonlinearity Measurement</u> .....	440
<u>Ammeter (Current Meter)</u> .....	441
<u>DAC, 8-Bit Differential Nonlinearity Measurement</u> .....	442
<u>DAC, 8-Bit Integral Nonlinearity Measurement</u> .....	443
<u>Delta Probe</u> .....	444
<u>Find Event Probe</u> .....	445

## Cadence Verilog-A Language Reference

---

<u>Find Slope</u>	447
<u>Frequency Meter</u>	448
<u>Offset Measurement</u>	449
<u>Power Meter</u>	450
<u>Q (Charge) Meter</u>	452
<u>Sampler</u>	453
<u>Slew Rate Measurement</u>	454
<u>Signal Statistics Probe</u>	455
<u>Voltage Meter</u>	457
<u>Z (Impedance) Meter</u>	458
<u>Mechanical Systems</u>	459
<u>Gearbox</u>	459
<u>Mechanical Damper</u>	460
<u>Mechanical Mass</u>	461
<u>Mechanical Restrainer</u>	462
<u>Road</u>	463
<u>Mechanical Spring</u>	464
<u>Wheel</u>	465
<u>Mixed-Signal Components</u>	466
<u>Analog-to-Digital Converter, 8-Bit</u>	466
<u>Analog-to-Digital Converter, 8-Bit (Ideal)</u>	467
<u>Decimator</u>	468
<u>Digital-to-Analog Converter, 8-Bit</u>	469
<u>Digital-to-Analog Converter, 8-Bit (Ideal)</u>	470
<u>Sigma-Delta Converter (first-order)</u>	471
<u>Sample-and-Hold Amplifier (Ideal)</u>	472
<u>Single Shot</u>	473
<u>Switched Capacitor Integrator</u>	474
<u>Power Electronics Components</u>	475
<u>Full Wave Rectifier, Two Phase</u>	475
<u>Half Wave Rectifier, Two Phase</u>	476
<u>Thyristor</u>	477
<u>Semiconductor Components</u>	478
<u>Diode</u>	478
<u>MOS Transistor (Level 1)</u>	479
<u>MOS Thin-Film Transistor</u>	481

## Cadence Verilog-A Language Reference

---

<u>N JFET Transistor</u> .....	482
<u>NPN Bipolar Junction Transistor</u> .....	483
<u>Schottky Diode</u> .....	485
<u>Telecommunications Components</u> .....	486
<u>AM Demodulator</u> .....	486
<u>AM Modulator</u> .....	487
<u>Attenuator</u> .....	488
<u>Audio Source</u> .....	489
<u>Bit Error Rate Calculator</u> .....	490
<u>Charge Pump</u> .....	491
<u>Code Generator, 2-Bit</u> .....	492
<u>Code Generator, 4-Bit</u> .....	493
<u>Decider</u> .....	494
<u>Digital Phase Locked Loop (PLL)</u> .....	495
<u>Digital Voltage-Controlled Oscillator</u> .....	496
<u>FM Demodulator</u> .....	497
<u>FM Modulator</u> .....	498
<u>Frequency-Phase Detector</u> .....	499
<u>Mixer</u> .....	500
<u>Noise Source</u> .....	501
<u>PCM Demodulator, 8-Bit</u> .....	502
<u>PCM Modulator, 8-Bit</u> .....	503
<u>Phase Detector</u> .....	504
<u>Phase Locked Loop</u> .....	505
<u>PM Demodulator</u> .....	506
<u>PM Modulator</u> .....	507
<u>QAM 16-ary Demodulator</u> .....	508
<u>Quadrature Amplitude 16-ary Modulator</u> .....	510
<u>QPSK Demodulator</u> .....	511
<u>QPSK Modulator</u> .....	512
<u>Random Bit Stream Generator</u> .....	513
<u>Transmission Channel</u> .....	514
<u>Voltage-Controlled Oscillator</u> .....	515



## 18

<u>Understanding Error Messages</u> .....	517
---	-----

## 19

<u>Getting Ready to Simulate</u> .....	519
--	-----

<u>Creating a Verilog-A Module Description</u> .....	520
--	-----

<u>File Extension .va</u> .....	520
---------------------------------	-----

<u>`include Compiler Directive</u> .....	520
--	-----

<u>CDS_MMSIM_VERILOGA Macro</u> .....	523
---------------------------------------	-----

<u>Creating a Spectre Netlist File</u> .....	523
--	-----

<u>Including Files in a Netlist</u> .....	524
---	-----

<u>Naming Requirements for SPICE-Mode Netlisting</u> .....	526
--	-----

<u>Modifying Absolute Tolerances</u> .....	527
--	-----

<u>Modifying abstol in Standalone Mode</u> .....	527
--	-----

<u>Modifying abstol in the Cadence Analog Design Environment</u> .....	529
--	-----

<u>Using the Compiled C Code Flow</u> .....	531
---	-----

<u>Compiling Verilog-A Compact Models for Reuse</u> .....	531
---	-----

<u>Reusing Verilog-A Shared Objects</u> .....	532
---	-----

<u>Controlling the Optimization Level of Compiled C Code Flow</u> .....	532
---	-----

<u>Turning the Compiled C Code Flow Off and On</u> .....	532
--	-----

<u>Creating and Specifying Compiled C Code Databases</u> .....	533
--	-----

<u>Reusing and Sharing Compiled C Objects</u> .....	534
---	-----

<u>Turning Off Parallel Compilation</u> .....	534
---	-----

<u>Using Verilog-A Compact Models to Increase Simulation Speed</u> .....	536
--	-----

<u>Noticing Differences When You Use the compact_module Attribute</u> .....	537
---	-----

<u>Specifying Instance and Model Parameters for a Verilog-A Compact Model</u> .....	537
---	-----

<u>Model Binning for Verilog-A Compact Models</u> .....	538
---	-----

<u>Ignoring the State of a Verilog-A Module for RF Simulation</u> .....	539
---	-----

<u>Ignoring the State of a Verilog-A Local Variable for RF Simulation</u> .....	540
---	-----

<u>Ignoring the States of all variables in a Verilog-A Module for RF Simulation</u> .....	542
---	-----

## 20

<u>Supported and Unsupported Language Elements</u> .....	545
--	-----

## 21

<u>Creating ViewInfo for Verilog-A Cellview</u> .....	549
---	-----

<u>ahdlUpdateViewInfo</u> .....	549
<u>Description</u> .....	549
<u>Arguments</u> .....	549
<u>Example 1</u> .....	549
<u>Example 2</u> .....	550
<u>Example 3</u> .....	550

## 22

<u>Converting SpectreHDL to Verilog-A</u> .....	551
---	-----

<u>SpectreHDL Constructs That Have Verilog-A Equivalents</u> .....	552
<u>SpectreHDL Constructs That Have No Verilog-A Equivalent</u> .....	557

## 23

<u>Verilog-A Source Protection</u> .....	559
--	-----

<u>Protecting the Source Description of Selected Modules or Regions</u> .....	561
<u>Using the Protection Pragmas</u> .....	562
<u>The nprotect Command</u> .....	563
<u>Protecting All Modules in a Source Description</u> .....	565

## 24

<u>Verilog-A Compliance</u> .....	567
-----------------------------------	-----

<u>Making Your Models Compliant</u> .....	568
<u>Analog Functions</u> .....	569
<u>NULL Statements</u> .....	569
<u>inf Used as a Number</u> .....	569
<u>Changing Delay to Absdelay</u> .....	570
<u>Changing \$realtime to \$abstime</u> .....	570

## Cadence Verilog-A Language Reference

---

<u>Changing bound_step to \$bound_step</u> .....	570
<u>Changing Array Specifications</u> .....	570
<u>Chained Assignments Made Illegal</u> .....	571
<u>Real Argument Not Supported as Direction Argument</u> .....	571
<u>\$limexp Changed to limexp</u> .....	571
<u>`if `MACRO is Not Allowed</u> .....	572
<u>discontinuity Changed to \$discontinuity</u> .....	572
<u>Noting Changes from OVI Verilog-AMS Version 2.0</u> .....	573
<u>Glossary</u> .....	575
<u>Index</u> .....	581

# Cadence Verilog-A Language Reference

---

# Preface

---

The Cadence® Verilog®-A language is the analog subset of the Verilog-AMS language. With Verilog-A, you can create and use modules that describe the high-level behavior of components and systems. You should first be familiar with the development, design, and simulation of circuits and with high-level programming languages, such as C.

See the following topics for additional information in this preface:

- [Related Documents](#) on page 21
- [Typographic and Syntax Conventions](#) on page 22

## Related Documents

For more information about Verilog-A and related products, consult the sources listed below.

- *Cadence Analog Design Environment User Guide*
- *Component Description Format User Guide*
- *Virtuoso Schematic Editor User Guide*
- *Cadence Hierarchy Editor User Guide*
- *Instance-Based View Switching Application Note*
- [\*Virtuoso Spectre Circuit Simulator Reference\*](#)
- [\*Virtuoso Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide\*](#)

## Typographic and Syntax Conventions

In general, the text in this book follows these typographic and syntax conventions:

<code>text</code>	Indicates text you must type exactly as it is presented.
<code>z_argument</code>	Indicates text that you must replace with an appropriate argument. The prefix (in this case, <code>z_</code> ) indicates the data type the argument can accept. Do not type the data type or underscore.
[ ]	Denotes an optional argument. When used with vertical bars, they enclose a list of choices from which you can choose one.
{ }	Used with vertical bars, they denote a list of choices from which you must choose one.
	Separates a choice of options.
...	Indicates that you can repeat the previous argument.
=>	Precedes the values returned by a Cadence® SKILL language function.
/	Separates the possible values that can be returned by a Cadence SKILL language function.
<i>text</i>	Indicates names of manuals, menu commands, form buttons, and form fields.

For other more specialized text, the following typographical conventions apply:

- The definition operator, `:=`, defines more complex elements of the Verilog-A language in terms of less complex elements.
- Lowercase words represent syntactic categories. For example,  
`module_declaration`  
`node_identifier`
- Boldface words represent elements of the syntax that must be used exactly as presented (except as noted below). Such items include keywords, operators, and punctuation marks. For example,  
**endmodule**

# Cadence Verilog-A Language Reference

## Preface

---

Sometimes options can be abbreviated. The shortest permitted abbreviation is shown by capital letters but you can use either upper or lower-case letters in your code. For example, the syntax

**-CHECKtasks**

means that you can type the option as `-checktasks`, `-CHECKTASKS`, `-ch`, `-CH`, `-cH`, and so on.

- Vertical bars indicate alternatives. You can choose to use any one of the items separated by the bars. For example,

```
attribute ::=
  abstol
  | access
  | ddt_nature
  | idt_nature
  | units
  | huge
  | blowup
  | identifier
```

- Square brackets enclose optional items. For example,

```
input declaration ::=
  input [ range ] list_of_port_identifiers ;
```

- Braces enclose an item that you can specify zero or more times. For example,

```
list_of_ports ::=
  ( port { , port } )
```

- Code examples appear in constant-width font.

```
/* This is an example of the font used for code.*/
```

- Within the text, variables are in italic font, like this: *allowed\_errors*.

- Keywords, file names, names of natures, and names of disciplines appear in constant-width font, like this:

```
keyword
file_name
name_of_nature
name_of_discipline
```

- If a statement is too long to fit on one line, the remainder of the statement appears indented on the next line, like this:

```
qgf = width*length*cfbb*(vgfs - wkf - qb/(2*cbb) -
      (vgbs - vfbb + qb/(2*cob))) + qgf_par ;
```

**Cadence Verilog-A Language Reference**  
Preface

---



---

# Modeling Concepts

---

This chapter introduces some important concepts basic to using the Cadence<sup>®</sup> Verilog<sup>®</sup>-A language, including:

- [Verilog-A Language Overview](#) on page 26
- [Describing a System](#) on page 27
- [Analog Systems](#) on page 28

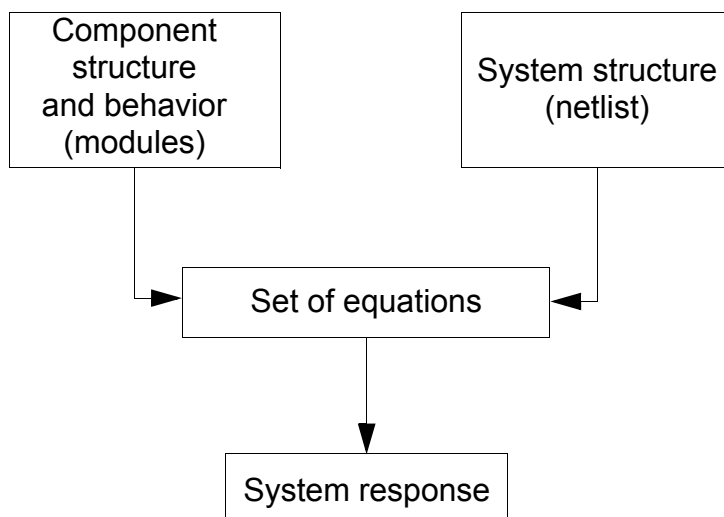
## Verilog-A Language Overview

The Verilog-A language is a high-level language that uses modules to describe the structure and behavior of analog systems and their components. With the analog statements of Verilog-A, you can describe a wide range of conservative systems and signal-flow systems, such as electrical, mechanical, fluid dynamic, and thermodynamic systems.

To describe a system, you must specify both the structure of the system and the behavior of its components. In Verilog-A with the Spectre<sup>®</sup> Circuit simulator, you define structure at different levels. At the highest level, you define overall system structure in a netlist. At lower, more specific levels, you define the internal structure of modules by defining the interconnections among submodules.

To specify the behavior of individual modules, you define mathematical relationships among their input and output signals.

After you define the structure and behavior of a system, the simulator derives a descriptive set of equations from the netlist and modules. The simulator then solves the set of equations to obtain the system response.



The simulator uses Kirchhoff's Potential and Flow laws to develop a set of descriptive equations and then solves the equations with the Newton-Raphson method. See [Appendix 15, "Nodal Analysis,"](#) for additional information.

To introduce the algorithms underlying system simulation, the following sections describe

- What a system is
- How you specify the structure and behavior of a system

- How the simulator develops a set of equations and solves them to simulate a system

## **Describing a System**

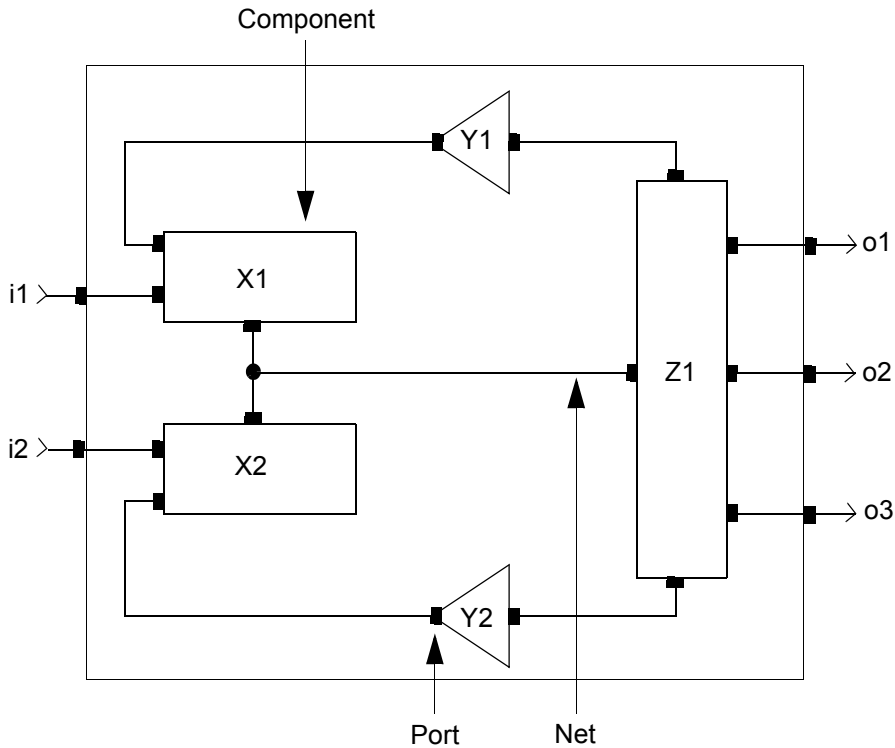
A *system* is a collection of interconnected components that produces a response when acted upon by a stimulus. A *hierarchical system* is a system in which the components are also systems. A *leaf component* is a component that has no subcomponents. Each leaf component connects to zero or more nets. Each net connects to a signal which can traverse multiple levels of the hierarchy. The behavior of each component is defined in terms of the values of the nets to which it connects.

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*.

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 with one analog connection and one

digital connection is a *mixed port*. The components interconnect through ports and nets to build a hierarchy, as illustrated in the following figure.

### System Terminology



## Analog Systems

The information in the following sections applies to analog systems such as the systems you can simulate with Verilog-A.

### Nodes

A node is a point of physical connection between nets of continuous-time descriptions. Nodes obey conservation-law semantics.

## Conservative Systems

A *conservative system* is one that obeys the laws of conservation described by Kirchhoff's Potential and Flow laws. For additional information about these laws, see "[Kirchhoff's Laws](#)" on page 324.

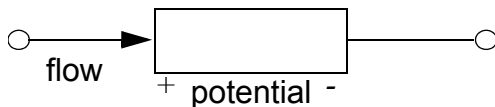
In a conservative system, each node has two values associated with it: the potential of the node and the flow out of the node. Each branch in a conservative system also has two associated values: the potential across the branch and the flow through the branch.

### Reference Nodes

The potential of a single node is defined with respect to a reference node. The reference node, called *ground* in electrical systems, has a potential of zero.

### Reference Directions

Each branch has a reference direction for the potential and flow. For example, consider the following schematic. With the reference direction shown, the potential in this schematic is positive whenever the potential of the terminal marked with a plus sign is larger than the potential of the terminal marked with a minus sign.



Verilog-A uses associated reference directions. Consequently, a positive flow is defined as one that enters the branch through the terminal marked with the plus sign and exits through the terminal marked with the minus sign.

## Signal-Flow Systems

Unlike conservative systems, signal-flow systems associate only a single value with each node. Verilog-A supports signal-flow modeling.

## Mixed Conservative and Signal-Flow Systems

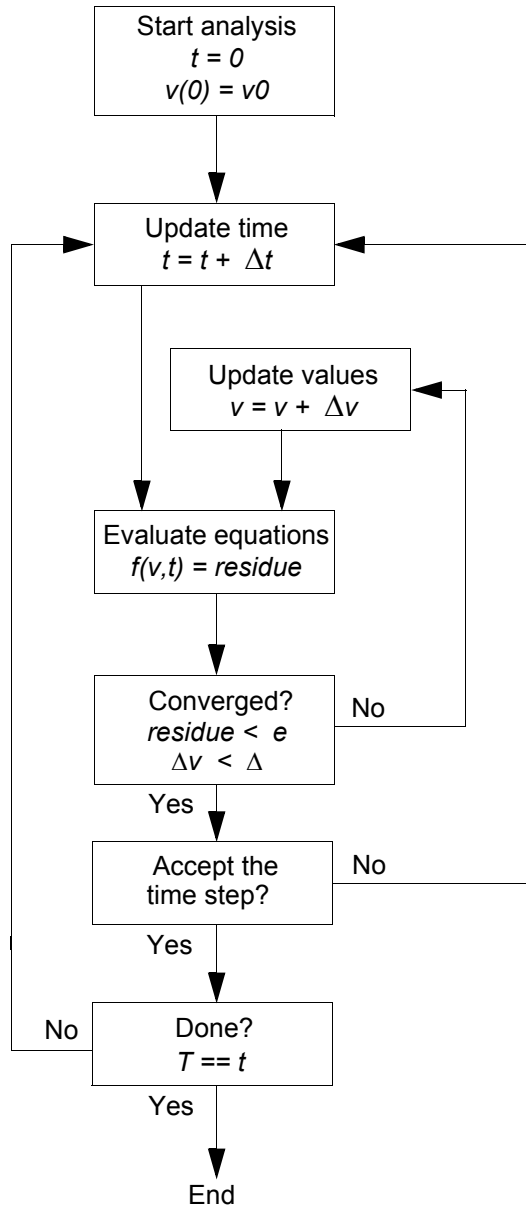
With Verilog-A, you can model systems that contain a mixture of conservative nodes and signal-flow nodes. Verilog-A accommodates this mixing with semantics that can be used for both kinds of nodes.

## Simulator Flow

After you specify the structure and behavior of a system, you submit the description to the simulator. The simulator then uses Kirchhoff's laws to develop equations that define the values and flows in the system. Because the equations are differential and nonlinear, the simulator does not solve them directly. Instead, the simulator uses an approximation and solves the equations iteratively at individual time points. The simulator controls the interval between the time points to ensure the accuracy of the approximation.

At each time point, iteration continues until two convergence criteria are satisfied. The first criterion requires that the approximate solution on this iteration be close to the accepted solution on the previous iteration. The second criterion requires that Kirchhoff's Flow Law be adequately satisfied. To indicate the required accuracy for these criteria, you specify tolerances. For a graphical representation of the analog iteration process, see the [Simulator Flow](#) figure on page 31. For more details about how the simulator uses Kirchhoff's laws, see "[Simulating a System](#)" on page 325.

### Simulator Flow



# Cadence Verilog-A Language Reference

## Modeling Concepts

---



---

## Creating Modules

---

This chapter describes how to use modules. The tasks involved in using modules are basic to modeling in Cadence® Verilog®-A.

- [Declaring Modules](#) on page 34
- [Declaring the Module Interface](#) on page 36
- [Defining Module Analog Behavior](#) on page 39
- [Using Internal Nodes in Modules](#) on page 43
- [Instantiating Modules with Netlists](#) on page 45

## Overview

This chapter introduces the concept of modules. Additional information about modules is located in [Chapter 10, “Instantiating Modules and Primitives.”](#) including detailed discussions about declaring and connecting ports and about instantiating modules.

The following definition for a digital to analog converter illustrates the form of a module definition. The entire module is enclosed between the keywords `module` and `endmodule` or `macromodule` and `endmodule`.

Interface declarations	{	<pre>module res1(p, n);   inout p, n;   electrical p, n;   parameter real r=1 from (0:inf);   parameter real tc=1.5m from [0:3m);</pre>
Behavioral description	{	<pre>    real reff;     analog begin       @(initial_step) begin         reff = r*(1+tc*\$temperature);       end       I(p, n) &lt;+ V(p, n)/reff ;     end endmodule</pre>

## Declaring Modules

To declare a module, use this syntax.

```
module_declaration ::=
  module_keyword module_identifier [ ( list_of_ports ) ] ;
  [ module_items ]
  endmodule |
```

```
module_declaration ::=
  module_keyword module_identifier [ ( list_of_port_declarations ) ] ;
  [ non_port_declarations_module_items ]
  endmodule
```

```
module_keyword ::=
  module
  | macromodule
```

```
module_items ::=
  { module_item }
  | analog_block
```

```
module_item ::=
  module_item_declaration
  | module_instantiation
```

## Cadence Verilog-A Language Reference

### Creating Modules

---

```
module_item_declaration ::=  
    parameter_declaration  
    | aliasparam_declaration  
    | input_declaration  
    | output_declaration  
    | inout_declaration  
    | ground_declaration  
    | integer_declaration  
    | net_discipline_declaration  
    | real_declaration
```

*module\_identifier*      The name of the module being declared.

*list\_of\_ports*              An ordered list of the module's ports. For details, see "[Ports](#)" on page 36.

*list\_of\_port\_declarations*  
    Declare the ports in the module header. For example,  

```
module A(input electrical a, output b);  
    electrical b;  
    .....  
endmodule
```

*module\_items*              The different types of declarations and definitions. Note that you can have no more than one analog block in each module.

---

#### For information about

#### Read

Analog blocks	<a href="#">"Defining Module Analog Behavior"</a> on page 39
Parameter overrides	<a href="#">"Overriding Parameter Values in Instances"</a> on page 220
Module instantiation	<a href="#">"Instantiating Verilog-A Modules"</a> on page 214
Parameter declarations	<a href="#">"Parameters and Local Parameters"</a> on page 58
Input, output, and inout declarations	<a href="#">"Port Direction"</a> on page 37
Integer declarations	<a href="#">"Units and descriptions specified for block-level variables are ignored by the simulator, but can be used for documentation purposes."</a> on page 56
Net discipline declarations	<a href="#">"Net Disciplines"</a> on page 75

<b>For information about</b>	<b>Read</b>
Real declarations	<a href="#">“Real Numbers”</a> on page 57
Genvar declarations	<a href="#">“Genvars”</a> on page 66
Analog function declarations	<a href="#">“User-Defined Functions”</a> on page 202

---

## Declaring the Module Interface

Use the module interface declarations to define

- Name of the module
- Ports of the module
- Parameters of the module

For example, the module interface declaration

```
module res(p, n) ;  
  inout p, n ;  
  electrical p, n ;  
  parameter real r = 0 ;
```

declares a module named `res`, ports named `p` and `n`, and a parameter named `r`.

### Module Name

To define the name for a module, put an identifier after the keyword `module` or `macromodule`. Ensure that the new module name is unique among other module, schematic, subcircuit, and model names, and any built-in Spectre<sup>®</sup> circuit simulator primitives. If your module has any ports, list them in parentheses following the identifier.

### Ports

To declare the ports used in a module, use port declarations. To specify the type and direction of a port, use the related declarations described in this section.

```
list_of_ports ::=  
  port { , port }  
list_of_port_declarations ::=  
  port_declaration { , port_declaration }  
port ::=  
  port_expression  
  
port_expression ::=
```

## Cadence Verilog-A Language Reference

### Creating Modules

---

```
    port_identifier
|   port_identifier [ constant_expression ]
|   port_identifier [ constant_range ]

constant_range ::=
    msb_constant_expression : lsb_constant_expression
port_declaration ::=
    inout [ discipline_identifier ] [ range ] [ port_identifier ]
|   input [ discipline_identifier ] [ range ] [ port_identifier ]
|   output [ discipline_identifier ] [ range ] [ port_identifier ]
```

For example, these code fragments illustrate possible port declarations.

```
module exam1 ;                // Defines no ports
module exam2 (p, n) ;        // Defines 2 simple ports
```

Normally, you cannot use `Q` as the name of a port. However, if you need to use `Q` as a port name, you can use the special text macro identifier, `VAMS_ELEC_DIS_ONLY`, as follows.

```
`define VAMS_ELEC_DIS_ONLY
`include "disciplines.vams"

(module 1, which uses a port called Q)
(module 2, which use a port called Q)
...
`include "disciplines.vams"

(module 3, which uses an access function called Q)
(module 4, which uses an access function called Q)
...
```

This macro undefines the sections in the `disciplines.vams` file that use `Q`, making it available for you to use as a port name. Consequently, when you need to use `Q` as an access function again, you need to include the `disciplines.vams` file again.

### Port Type

To declare the type of a port, use a net discipline declaration in the body of the module. If you do not declare the type of a port, you can use the port only in a structural description. In other words, you can pass the port to module instances, but you cannot access the port in a behavioral description. Net discipline declarations are described in [“Net Disciplines”](#) on page 75.

Ports declared as vectors must use identical ranges for the port type and port direction declarations.

### Port Direction

You must declare the port direction for every port in the list of ports section of the module declaration. To declare the direction of a port, use one of the following three syntaxes.

## Cadence Verilog-A Language Reference

### Creating Modules

---

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

output_declaration ::=
    output [ range ] list_of_port_identifiers ;

inout_declaration ::=
    inout [ range ] list_of_port_identifiers ;

range ::=
    [ constant_expression : constant_expression ]
```

<code>input</code>	Declares that the signals on the port cannot be set, although they can be used in expressions.
<code>output</code>	Declares that the signals on the port can be set, but they cannot be used in expressions.
<code>inout</code>	Declares that the port is bidirectional. The signals on the port can be both set and used in expressions. <code>inout</code> is the default port direction.

Ports declared as vectors must use identical ranges for the port type and port direction declarations.

In this release of Verilog-A, the compiler does not enforce correct application of `input`, `output`, and `inout`.

### Port Declaration Example

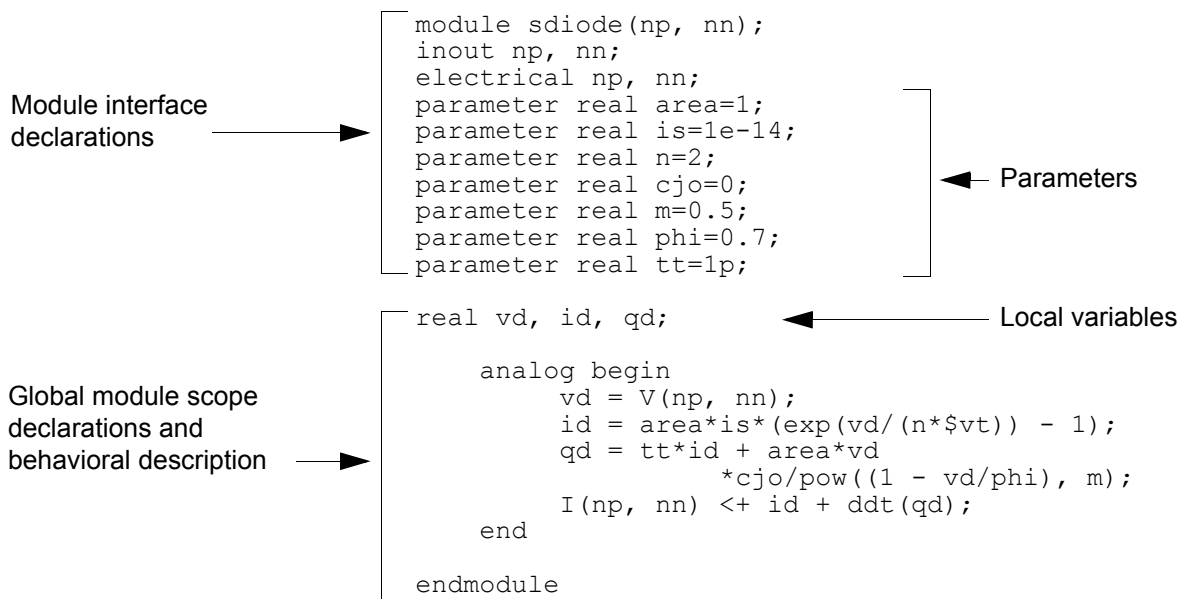
Module `gainer`, described below, has two ports: `out` and `pin`. The `out` port is declared with a port direction of `output`, so that its values can be set. The `pin` port is declared with a port direction of `input`, so that its value can be read. Both ports are declared to be of the `voltage` discipline.

```
module gainer (out, pin) ;           // Declares two ports
output out ;                         // Declares port as output
input pin ;                           // Declares port as input
voltage out, pin ;                   // Declares type of ports
parameter real gain = 2.0 ;
analog
    V(out) <+ gain * V(pin) ;
endmodule
```

## Parameters

With parameter (and dynamicparam) declarations, you specify parameters that can be changed when a module is used as an instance in a design. Using parameters lets you customize each instance.

For each parameter, you must specify a default value. You can also specify an optional type and an optional valid range. The following example illustrates how to declare parameters and variables in a module.



Module `sdiode` has a parameter, `area`, that defaults to 1. If `area` is not specified for an instance, it receives a value of 1. Similarly, the other parameters, `is`, `n`, `cjo`, `m`, `phi`, and `tt`, have specified default values too.

Module `sdiode` also defines three local variables: `vd`, `id`, and `qd`.

For more information about parameter declarations, see [“Parameters and Local Parameters”](#) on page 58.

## Defining Module Analog Behavior

To define the behavioral characteristics of a module, you create an analog block. The simulator evaluates all the analog blocks in the various modules of a design as though the blocks are executing concurrently.

## Cadence Verilog-A Language Reference

### Creating Modules

---

```
analog_block ::=
    analog analog_statement

analog_statement ::=
    analog_seq_block
    | analog_branch_contribution
    | analog_indirect_branch_assignment
    | analog_procedural_assignment
    | analog_conditional_statement
    | analog_for_statement
    | analog_case_statement
    | analog_event_controlled_statement
    | system_task_enable
```

`analog_statement` can appear only within the analog block.

`analog_seq_block` are discussed in [“Sequential Block Statement”](#) on page 86.

In the analog block, you can code contribution statements that define relationships among analog signals in the module. For example, consider the following contribution statements:

```
V(n1, n2) <+ expression;
I(n1, n2) <+ expression;
```

where  $V(n1, n2)$  and  $I(n1, n2)$  represent potential and flow sources, respectively. You can define `expression` to be any combination of linear, nonlinear, algebraic, or differential expressions involving module signals, constants, and parameters.

The modules you write can contain at most a single analog block. When you use an analog block, you must place it after the interface declarations and local declarations.

The following module, which produces the sum and product of its inputs, illustrates the form of the analog block. Here the block contains two contribution statements.

```
module am(in1, in2, outsum, outmult) ;
input in1, in2 ;
output outsum, outmult ;
voltage in1, in2, outsum, outmult ;

    analog begin
        V(outsum) <+ V(in1) + V(in2) ;
        V(outmult) <+ V(in1) * V(in2) ;
    end
endmodule
```

Module `setvolts` illustrates an analog block containing a single statement.

```
module setvolts (outvolt) ;
output outvolt ;
voltage outvolt ;

    analog
        V(outvolt) <+ 5.0 ;

endmodule
```

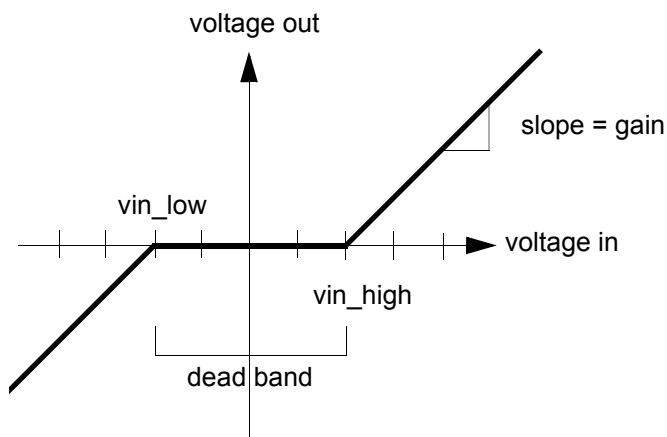


**Note:** You can define multiple analog blocks within a module description. When multiple analog blocks are defined, the simulator concatenates them internally in the order that they are defined within the module description, and treats them as one analog block. In other words, the analog blocks are executed in the order that they are specified in the module. Concatenation of the analog blocks occurs after all `genvar` constructs have been evaluated, that is, after the `genvar` constructs have been unrolled and the conditional `genvar` constructs have been selected. If an analog block appears in a `genvar` statement loop, the order in which the loop is unrolled during elaboration determines the order in which the analog blocks are concatenated into one analog block after elaboration. Concurrent evaluation of multiple analog blocks within a module has similar behavior when compared to the evaluation of a single analog block that was created by concatenating multiple analog blocks within the same module.

## Defining Analog Behavior with Control Flow

You can also incorporate conditional control flow into a module. With control flow, you can define the behavior of a module in regions.

The following module, for example, describes a voltage deadband amplifier `vdba`. If the input voltage is greater than `vin_high` or less than `vin_low`, the amplifier is active. When the amplifier is active, the output is `gain` times the differential voltage between the input voltage and the edge of the deadband. When the input is in the deadband between `vin_low` and `vin_high`, the amplifier is quiescent and the output voltage is zero.



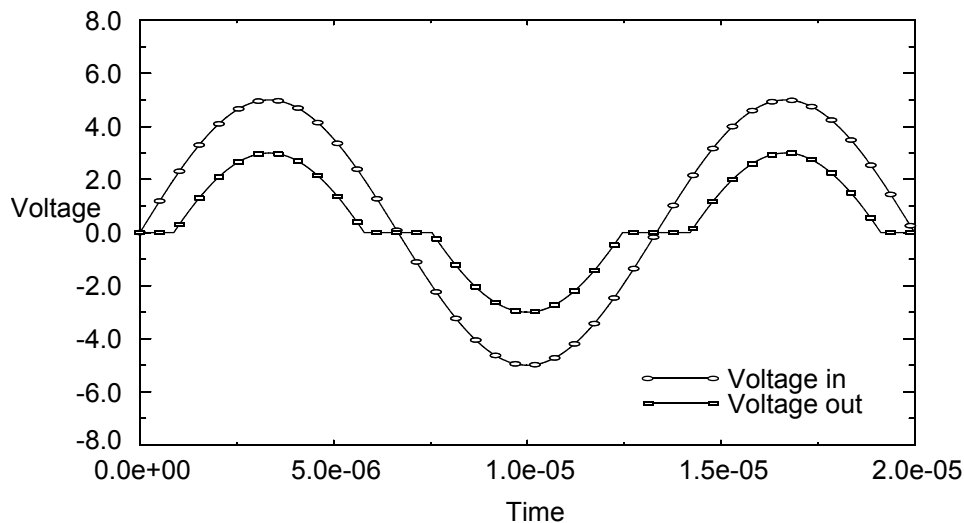
```
module vdba(in, out);
input in ;
output out ;
electrical in, out ;
parameter real vin_low = -2.0 ;
parameter real vin_high = 2.0 ;
parameter real gain = 1 from (0:inf) ;
    analog begin
        if (V(in) >= vin_high) begin
```

## Cadence Verilog-A Language Reference

### Creating Modules

```
        V(out) <+ gain*(V(in) - vin_high) ;
    end else if (V(in) <= vin_low) begin
        V(out) <+ gain*(V(in) - vin_low) ;
    end else begin
        V(out) <+ 0 ;
    end
end
endmodule
```

The following graph shows the response of the `vdba` module to a sinusoidal source.



## Using Integration and Differentiation with Analog Signals

The relationships that you define among analog signals can include time domain differentiation and integration. Verilog-A provides a time derivative function, `ddt`, and two time integral functions, `idt` and `idtmod`, that you can use to define such relationships. For example, you might write a behavioral description for an inductor as follows.

```
module induc(p, n);
  inout p, n;
  electrical p, n;
  parameter real L = 0;
  analog
    V(p, n) <+ ddt(L * I(p, n)) ;
endmodule
```

In module `induc`, the voltage across the external ports of the component is defined as equal to the time derivative of  $L$  times the current flowing between the ports.

To define a higher order derivative, you must use an internal node or signal. For example, module `diff_2` defines internal node `diff`, and sets  $V(\text{diff})$  equal to the derivative of

$V(in)$ . Then the module sets  $V(out)$  equal to the derivative of  $V(diff)$ , in effect taking the second order derivative of  $V(in)$ .

```
module diff_2(in, out) ;
input in ;
output out ;
electrical in, out ;
electrical diff ;    // Defines an internal node.
    analog begin
        V(diff) <+ ddt(V(in)) ;
        V(out) <+ ddt(V(diff)) ;
    end
endmodule
```

For time domain integration, use the `idt` or `idtmod` functions, as illustrated in module `integrator`.

```
module integrator(in, out) ;
input in ;
output out ;
electrical in, out ;

    analog begin
        V(out) <+ idt(V(in), 0) ;
    end
endmodule
```

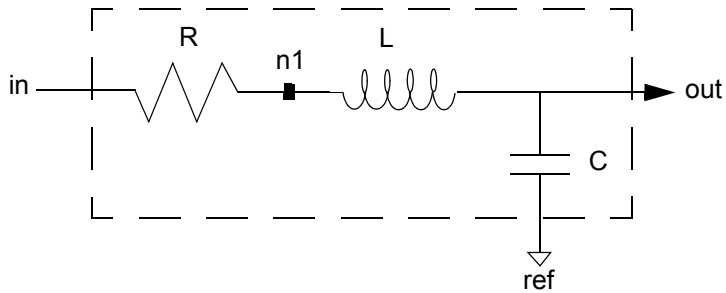
Module `integrator` sets the output voltage to the integral of the input voltage. The second term in the `idt` function is the initial condition. For more information on `ddt`, `idtmod`, and `idt`, refer to [“Time Derivative Operator”](#) on page 162, [“Circular Integrator Operator”](#) on page 164, and [“Time Integral Operator”](#) on page 162.

## Using Internal Nodes in Modules

Using Verilog-A, you can implement complex designs in a variety of different ways. For example, you can define behavior in modules at the leaf level and use the netlist to define the structure of the system. You can also define structure within modules by defining internal nodes. With internal nodes, you can directly define behavior in the module, or you can introduce internal nodes as a means of solving higher order differential equations that define the network.

## Using Internal Nodes in Behavioral Definitions

Consider the following RLC circuit.



Module `rlc_behav` uses an internal node `n1` and the ports `in`, `ref`, and `out`, to define directly the behavioral characteristics of the RLC circuit. Notice how `n1` does not appear in the list of ports for the module.

```
module rlc_behav(in, out, ref) ;
  inout in, out, ref ;
  electrical in, out, ref ;
  parameter real R=1, L=1, C=1 ;

  electrical n1 ;

  analog begin
    V(in, n1) <+ R*I(in, n1) ;
    V(n1, out) <+ L*ddt(I(n1, out)) ;
    I(out, ref) <+ C*ddt(V(out, ref)) ;
  end
endmodule
```

## Using Internal Nodes in Higher Order Systems

You can also represent the RLC circuit by its governing differential equations. The transfer function is given by

$$H(s) = \frac{1}{LCs^2 + RCs + 1} = \frac{V_{out}}{V_{in}}$$

In the time domain, this becomes

$$V_{out} = V_{in} - R \cdot C \cdot \dot{V}_{out} - L \cdot C \cdot \ddot{V}_{out}$$

If you set

$$V_{n1} = \dot{V}_{out}$$

you can write

$$V_{out} = V_{in} - R \cdot C \cdot V_{n1} - L \cdot C \cdot \dot{V}_{n1}$$

Module `rlc_high_order` implements these descriptions.

```
module rlc_high_order(in, out, ref) ;
inout in, out, ref ;
electrical in, out, ref ;
parameter real R=1, L=1, C=1 ;

    electrical n1 ;

    analog begin
        V(n1, ref) <+ ddt(V(out, ref)) ;
        V(out, ref) <+ V(in) - (R*C*V(n1) - L*ddt(V(n1))*C) ;
    end
endmodule
```

## Instantiating Modules with Netlists

After you define your Verilog-A modules, you can use them as ordinary primitives in other modules and in Spectre. For information on instantiating modules in netlists, see [Appendix 19, “Getting Ready to Simulate.”](#) For additional information about simulating, and for information specifically tailored for using Verilog-A in the Cadence analog design environment, see [Chapter 13, “Using Verilog-A in the Cadence Analog Design Environment.”](#)

**Cadence Verilog-A Language Reference**  
Creating Modules

---

---

## Lexical Conventions

---

A Cadence® Verilog®-A source text file is a stream of lexical tokens arranged in free format. For information, see, in this chapter,

- [White Space](#) on page 48
- [Comments](#) on page 48
- [Identifiers](#) on page 48
- [Numbers](#) on page 50

See also

- [Operators for Analog Blocks](#) on page 93
- The information about strings in [Displaying Results](#) on page 183

## White Space

White space consists of blanks, tabs, new-line characters, and form feeds. Verilog-A ignores these characters except in strings or when they separate other tokens. For example, this code fragment

```
$strobe("bit error rate = %f%%",  
      100.0 * errors / bits ) ;
```

is syntactically identical to:

```
$strobe("bit error rate = %f%%",100.0*errors/bits);
```

## Comments

In Verilog-A, you can designate a comment in either of two ways.

- A one-line comment starts with the two characters `//` (provided they are not part of a string) and ends with a new-line character. Within a one-line comment, the characters `/`, `/*`, and `*/` have no special meaning. A one-line comment can begin anywhere in the line.

```
//  
// This code fragment contains four one-line comments.  
parameter real vos ; // vos is the offset voltage  
//
```

- A block comment starts with the two characters `/*` (provided they are not part of a string) and ends with the two characters `*/`. Within a block comment, the characters `/*` and `/` have no special meaning.

```
/*  
* This is an example of a block comment. A block  
comment can continue over several lines, making it  
easy to add extended comments to your code.  
*/
```

## Identifiers

You use an identifier to give a unique name to an object, such as a variable declaration or a module, so that the object can be referenced from other places. There are two kinds of identifiers: *ordinary identifiers* and *escaped names*. Both kinds are case sensitive.



## Ordinary Identifiers

The first character of an ordinary identifier must be a letter or an underscore character (`_`), but the remaining characters can be any sequence of letters, digits, dollar signs (`$`), and the underscore. Examples include

```
unity_gain_bandwidth
holdValue
HoldTime
_bus$2
```

## Escaped Names

Escaped names start with the backslash character (`\`) and end with white space. Neither the backslash character nor the terminating white space is part of the identifier. Therefore, the escaped name `\pin2` is the same as the ordinary identifier `pin2`.

An escaped name can include any of the printable ASCII characters (the decimal values 33 through 126 or the hexadecimal values 21 through 7E). Examples of escaped names include

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

**Note:** The Spectre<sup>®</sup> Circuit simulator netlist does not recognize names escaped in this way. In Spectre, characters are individually escaped so that `\!!!error_condition!!!` is referred to as `\!\!\!error_condition\!\!\!` in the Spectre netlist.

## Scope Rules

In Verilog-A, each module, task, function, analog function, and named block that you define creates a new scope. Within a scope, an identifier can declare only one item. This rule means that within a scope you cannot declare two variables with the same name, nor can you give an instance the same name as a node connecting that instance.

Any object referenced from a named block must be declared in one of the following places.

- Within the named block
- Within a named block or module that is higher in the branch of the name tree

To find a referenced object, the simulator first searches the local scope. If the referenced object is not found in the local scope, the simulator moves up the name tree, searching

through containing named blocks until the object is found or the module boundary is reached. If the module boundary is reached before the object is found, the simulator issues an error.

## Numbers

Verilog-A supports two basic literal data types for arithmetic operations: *integer numbers* and *real numbers*.

### Integer Numbers

The syntax for an integer constant is

```
integer_number ::=
    [ sign ] decimal_num
    |[ sign ] octal_num
    |[ sign ] binary_num
    |[ sign ] hex_num

sign ::=
    + | -

decimal_number ::=
    [ unsigned_num ]
    |[size ] decimal_base unsign_num
binary_num ::=
    [ size] binary_base binary_val
octal_num ::=
    [ size ] octal_base octal_val
hex_num ::=
    [ size ] octal_base octal_val
size ::=
    non_zero_unsign_num
non_zero_unsign_num ::=
    non_zero_decimal_digit { _ | decimal_digit }
unsign_num ::=
    decimal_digit { _ | decimal_digit }
binary_val ::=
    binary_digit { _ | binary_digit }
octal_val ::=
    octal_digit { _ | octal_digit }
hex_val ::=
    hex_digit { _ | hex_digit }
decimal_base ::=
    '[s]d | '[s]D
binary_base ::=
    '[s]b | '[s]B
octal_base ::=
    '[s]o | '[s]O
hex_base ::=
    '[s]h | '[s]H
non_zero_decimal_digit ::=
    1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
decimal_digit ::=
```

## Cadence Verilog-A Language Reference

### Lexical Conventions

---

```
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
binary_digit ::=
    0 | 1
octal_digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
hex_digit ::=
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    | a | b | c | d | e | f | A | B | C | D | E | F
```

An integer constant can be specified in decimal, hexadecimal, octal, or binary format and can be expressed in the following two forms:

- A simple decimal number written as a sequence of digits from 0 to 9. The number may also start with a + or - unary operator.
- A base constant comprising of up to three tokens.

(Optional) The first token specifies the size of the constant in terms of number of bits. It is specified as a non-zero unsigned decimal number. For example, the size of a hexadecimal digit is 4.

The second token is a base format consisting of the following case-insensitive letters specifying the base for the number:

- d or D - for decimal
- h or H - for hexadecimal
- o or O - for octal
- b or B - for binary

These letters may be preceded by the single character s (or S) to indicate a signed quantity, and by the apostrophe (') character. The apostrophe character and the base format character should not contain any space between them.

The third token is an unsigned number comprising of legal digits for the specified base format. The number immediately follows the base format and may be preceded by a white space. The hexadecimal digits a to f are case insensitive.

Simple decimal numbers not having the size and base format, are considered as signed integers. However, if the numbers are specified with the base format and also include the s (or S) character, they are called signed numbers. Numbers having only the base format are considered as unsigned numbers.

The s (or S) character does not affect the specified bit pattern, but affects only its interpretation. A + or - operator preceding the size constant is a unary + or - operator. A + or - operator between the base format and the number is an illegal syntax. Negative numbers are represented in 2's complement form.

## Cadence Verilog-A Language Reference

### Lexical Conventions

---

If the size of the unsigned number is smaller than the size specified for the constant, zeros are added to the left of the unsigned number. If the size of the unsigned number is larger than the size specified for the constant, the unsigned number is truncated from the left. The number of bits that make up an unsigned number (which is a simple decimal number or a number without the size specification) is at least 32.

The simulator ignores the underscore character (`_`), so you can use it anywhere in a decimal number except as the first character. Using the underscore character can make long numbers more legible.

The following are examples of integer constants:

#### Unsigned constant numbers

```
568 // a decimal number
'h 325FF // a hexadecimal number
'o2530 // an octal number
laf // illegal (hexadecimal format requires 'h)
```

#### Sized constant numbers

```
3'b1001 // is a 3-bit binary number
4'D 2 // is a 4-bit decimal number
3'b0101 // is truncated to a 3-bit binary number
11'h1a // is padded to a 11-bit hexadecimal number
```

#### Signed constant numbers

```
3 'd -3 // illegal syntax
-6 'd 4 // defines the two's complement of 4, held in 6 bits-equivalent to -(6'd 4)
```

#### Underscore characters in numbers

```
15_145_000
12'b0011_1111
16 'h 10ab_f001
```

## Real Numbers

The syntax for a real constant is

```
real_number ::=
    [ sign ] unsign_num .unsign_num
    | [ sign ] unsign_num [ .unsign_num ] e [ sign ] unsign_num
    | [ sign ] unsign_num [ .unsign_num ] E [ sign ] unsign_num
    | [ sign ] unsign_num [ .unsign_num ] unit_letter

sign ::=
    + | -

unsign_num ::=
    decimal_digit { _ | decimal_digit }
```

## Cadence Verilog-A Language Reference

### Lexical Conventions

---

```
decimal_digit ::=  
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9  
unit_letter ::=  
    T | G | M | K | k | m | u | n | p | f | a
```

`unit_letter` represents one of the scale factors listed in the following table. If you use `unit_letter`, you must not have any white space between the number and the letter. Be certain that you use the correct case for the `unit_letter`.

---

<code>unit_letter</code>	Scale factor	<code>unit_letter</code>	Scale factor
T =	$10^{12}$	k =	$10^3$
G =	$10^9$	m =	$10^{-3}$
M =	$10^6$	u =	$10^{-6}$
K =	$10^3$	n =	$10^{-9}$
		p =	$10^{-12}$
		f =	$10^{-15}$
		a =	$10^{-18}$

---

The simulator ignores the underscore character ( `_` ), so you can use it anywhere in a real number except as the first character. Using the underscore character can make long numbers more legible.

Examples of real constants include

```
2.5K           // 2500  
1e-6           // 0.000001  
-9.6e9  
-1e-4  
0.1u  
50p           // 50 * 10e-13  
1.2G           // 1.2 * 10e8  
213_116.223_642
```

For information on converting real numbers to integer numbers, see [“Converting Real Numbers to Integer Numbers”](#) on page 57.

# Cadence Verilog-A Language Reference

## Lexical Conventions

---

---

## Data Types and Objects

---

The Cadence® Verilog®-A language defines these data types and objects. For information about how to use them, see the indicated locations.

- Units and descriptions specified for block-level variables are ignored by the simulator, but can be used for documentation purposes. on page 56
- Real Numbers on page 57
- Strings on page 58
- Parameters and Local Parameters on page 58
- String Parameters on page 63
- Parameter Aliases on page 64
- Paramsets on page 64
- Genvars on page 66
- Natures on page 68
- Disciplines on page 71
- Net Disciplines on page 75
- Named Branches on page 77
- Implicit Branches on page 78
- Output Variables on page 79
- Digital Nets and Registers

## Output Variables

The standard attributes for descriptions and units, have a special meaning for variables declared at module scope. Module scope variables with a description or units attribute, or both, are known as output variables and Cadence tools provide access to their values. Also Cadence tools print the names, values, units, and descriptions of output variables for primitives when displaying operating-point information.

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 Cadence tools include the following information:

```
cgs=4.21e-15 F
```

Descriptions for instance parameters of `mos_inst`:

```
cgs: gate-source capacitance
```

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

## Integer Numbers

Use the `integer` declaration to declare variables of type integer.

```
integer_declaration ::=  
    integer list_of_identifiers ;  
list_of_identifiers ::=  
    var_name { , var_name }  
var_name ::=  
    variable_identifier  
    | array_identifier [ range ]  
range ::=  
    upper_limit_const_exp : lower_limit_const_exp
```

In Verilog-A, you can declare an integer number in a range at least as great as  $-2^{31}$  ( $-2,147,483,648$ ) to  $2^{31}-1$  ( $2,147,483,647$ ).

To declare an array, specify the upper and lower indexes of the range. Be sure that each index is a constant expression that evaluates to an integer value.



## Cadence Verilog-A Language Reference

### Data Types and Objects

---

```
integer a[1:64] ;           // Declares array of 64 integers
integer b, c, d[-20:0] ;   // Declares 2 integers and an array
parameter integer max_size = 15 from [1:50] ;
integer cur_vector[1:max_size] ;
/* If the max_size parameter is not overridden, the
previous two statements declare an array of 15 integers. */
```

The standard attributes for descriptions and units can be used with integer declarations. For example,

```
(* desc="index number", units="index" *) integer indx;
```

## Real Numbers

Use the `real` declaration to declare variables of type `real`.

```
real_declaration ::=
    real list_of_identifiers ;
list_of_identifiers ::=
    var_name { , var_name }
var_name ::=
    variable_identifier
    | array_identifier [ range ]
range ::=
    upper_limit_const_exp : lower_limit_const_exp
```

In Verilog-A, you can declare real numbers in a range at least as great as  $10^{-37}$  to  $10^{+37}$ . To declare an array of real numbers, specify the upper and lower indexes of the range. Be sure that each index is a constant expression that evaluates to an integer value.

```
real a[1:64] ;           // Declares array of 64 reals
real b, c, d[-20:0] ;   // Declares 2 reals and an array of reals
parameter integer min_size = 1, max_size = 30 ;
real cur_vector[min_size:max_size] ;
/* If the two parameters are not overridden, the
previous two statements declare an array of 30 reals. */
```

Real variables have default initial values of zero.

The standard attributes for descriptions and units can be used with real declarations. For example,

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

## Converting Real Numbers to Integer Numbers

Verilog-A converts a real number to an integer number by rounding the real number to the nearest integer. If the real number is equally distant from the two nearest integers, Verilog-A

converts the real number to the integer farthest from zero. The following code fragment illustrates what happens when real numbers are assigned to integer numbers.

```
integer    intvalA, intvalB, intvalC ;
real      realvalA, realvalB, realvalC ;

realvalA = -1.7 ;
intvalA = realvalA ; // intvalA is -2

realvalB = 1.5 ;
intvalB = realvalB ; // intvalB is 2

realvalC = -1.5 ;
intvalC = realvalC ; // intvalC is -2
```

If either operand in an expression is real, Verilog-A converts the other operand to real before applying the operator. This conversion process can result in a loss of information.

```
real realvar ;
realvar = 9.0 ;
realvar = 4/3 * realvar ; // realvar is 9.0, not 12.0
```

In this example, both 4 and 3 are integers, so 1 is the result of the division. Verilog-A converts 1 to 1.0 before multiplying the converted number by 9.0.

## Strings

Use the `string` declaration to declare variables of type string.

```
string_declaration ::=
    string list_of_identifiers ;

list_of_identifiers ::=
    variable_identifier { , variable_identifier }
```

A string is defined as follows:

```
string ::=
    " { Any_ASCII_character_except_newline } "
```

For example,

```
string tmpString, difString;
tmpString="Temporary string";
difString="Different string";
```

## Parameters and Local Parameters

Use the `parameter` declaration to specify the parameters of a module.

```
parameter_declaration ::=
    parameter [opt_type] list_of_param_assignments ;
```

## Cadence Verilog-A Language Reference

### Data Types and Objects

---

Use the `localparam` declaration to specify local parameters for a module.

```
local_parameter_declaration ::=  
    localparam [opt_type] list_of_param_assignments ;
```

**Note:** Local parameters are identical to parameters except that you cannot modify them directly using an ordered or named parameter value assignment. Instead, you can assign a local parameter to a constant expression containing a parameter that you can modify with an ordered or named parameter value assignment.

```
opt_type ::=  
    real  
    | integer  
    | string  
  
list_of_param_assignments ::=  
    declarator_init {, declarator_init }  
  
declarator_init ::=  
    parameter_id = constant_expression { opt_value_range }  
    | parameter_array_init
```

For information about `opt_type`, see [“Specifying a Parameter Type”](#) on page 61. Note that for parameter arrays, however, you must specify a type.

For information about `opt_value_range`, see [“Specifying Permissible Values”](#) on page 61.

`parameter_id` is the name of a parameter you are declaring.

For information about `parameter_array_init`, see [“Specifying Parameter Arrays”](#) on page 62.

As specified in the syntax, the right-hand side of each `declarator_init` assignment must be a constant expression. You can include in the constant expression only constant numbers and previously defined parameters.

Parameters are constants, so you cannot change the value of a parameter at runtime. However, you can customize module instances by changing parameter values during compilation. See [“Overriding Parameter Values in Instances”](#) on page 220 for more information.

Consider the following code fragment. The parameter `superior` is defined by a constant expression that includes the parameter `subord`.

```
parameter integer subord = 8 ;  
parameter integer superior = 3 * subord ;
```

In this example, changing the value of `subord` changes the value of `superior` too because the value of `superior` depends on the value of `subord`.

The standard attributes for descriptions and units can be used with parameter declarations. For example,

## Cadence Verilog-A Language Reference

### Data Types and Objects

---

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

The attribute for inherited parameters, (`* cds_inherited_parameter *`), can also be used with parameter declarations (and only with parameter declarations) to obtain parameter values directly from the hierarchy where the module is instantiated. This attribute enables Monte Carlo mismatch for Verilog-A devices.

The inherited parameter attribute is subject to the following requirements:

- The parameter that is to be inherited must be defined in the hierarchy.
- The type of the parameter must be real. Integer and string parameters cannot be inherited.
- The inherited parameter must be initialized to a value of zero.
- The value of an inherited parameter must not be changed by the instance statement for the module. However, an ordinary parameter whose value is set by referring to an inherited parameter can be changed by the instance statement.

For example, to run the `ahdlLib.res` cell in Monte Carlo, you modify the Verilog-A model to be something like this:

```
module res(vp, vn);
inout vp, vn;
electrical vp, vn;
(* cds_inherited_parameter *) parameter real monteres = 0;
parameter real r = 1k;
localparam real r_effective = r + monteres; // nominal resistance plus
// monte-carlo mismatch effect
analog
    V(vp, vn) <+ (r_effective)*I(vp, vn);
endmodule
```

In this case, `monteres` is the mismatch parameter. It must be defined in a model deck as a `parameters` statement or be defined in the design variables section of the user interface.

You also need a `statistics mismatch` block in your model deck that describes the distribution for `monteres`. For example:

```
parameters monteres=10
statistics {
    mismatch {
        vary monteres dist=gauss std=5
    }
}
```

## Specifying a Parameter Type

You must specify a default for each parameter you define, but the parameter type specifier is optional (except that you must specify a type for parameter arrays). If you omit the parameter type specifier, Verilog-A determines the parameter type from the constant expression. If you do specify a type, and it conflicts with the type of the constant expression, your specified type takes precedence.

The three parameter declarations in the following examples all have the same effect. The first example illustrates a case where the type of the expression agrees with the type specified for the parameter.

```
parameter integer rate = 13 ;
```

The second example omits the parameter type, so Verilog-A derives it from the integer type of the expression.

```
parameter rate = 13 ;
```

In the third example, the expression type is real, which conflicts with the specified parameter type. The specified type, integer, takes precedence.

```
parameter integer rate = 13.0
```

In all three cases, `rate` is declared as an integer parameter with the value 13.

## Specifying Permissible Values

Use the optional range specification to designate permissible values for a parameter. If you need to, you can specify more than one range.

```
opt_value_range ::=
    from value_range_specifier
    | exclude value_range_specifier
    | exclude value_constant_expression
value_range_specifier ::=
    start_paren expression1 : expression2 end_paren
start_paren ::=
    [
    | (
end_paren ::=
    ]
    | )
expression1 ::=
    constant_expression | -inf
expression2 ::=
    constant_expression | inf
```

Ensure that the first expression in each range specifier is smaller than the second expression. Use a bracket, either “[” for the lower bound or “]” for the upper, to include an end point in the range. Use a parenthesis, either “(” for the lower bound or “)” for the upper, to exclude an end point from the range. To indicate the value infinity in a range, use the keyword `inf`. To indicate negative infinity, use `-inf`.

For example, the following declaration gives the parameter `cur_val` the default of -15.0. The range specification allows `cur_val` to acquire values in the range  $-\infty < \text{cur\_val} < 0$ .

```
parameter real maxval = 0.0 ;
parameter real cur_val = -15.0 from (-inf:maxval) ;
```

The following declaration

```
parameter integer pos_val = 30 from (0:40] ;
```

gives the parameter `pos_val` the default of 30. The range specification for `pos_val` allows it to acquire values in the range  $0 < \text{pos\_val} \leq 40$ .

In addition to defining a range of permissible values for a parameter, you can use the keyword `exclude` to define certain values as illegal.

```
parameter low = 10 ;
parameter high = 20 ;
parameter integer intval = 0 from [0:inf) exclude (low:high] exclude 5 ;
```

In this example, both a range of values,  $10 < \text{value} \leq 20$ , and the single value 5 are defined as illegal for the parameter `intval`.

## Specifying Parameter Arrays

Use the parameter array initiation part of the `parameter` or `localparam` declaration (`parameter_array_init`) to specify information for parameter arrays.

```
parameter_array_init ::=
    parameter_array_id range = constant_param_arrayinit {opt_value_range}
range ::=
    [ constant_expression : constant_expression ]
constant_param_arrayinit ::=
    { param_arrayinit_element_list }
    | '{ param_arrayinit_element_list }
    | '{ replicator_element_list }
param_arrayinit_element_list ::=
    constant_expression { , constant_expression }
replicator_element_list ::=
    | replicator_constant_expression {constant_expression}
```

`parameter_array_id` is the name of a parameter array you are declaring.

For information about `opt_value_range`, see [“Specifying Permissible Values”](#) on page 61.

`replicator_constant_expression` is an integer constant with a value greater than zero that specifies the number of times you want to include the associated `constant_expression` in the element list.

For example, you might declare and use a parameter array as follows:

```
parameter integer
  IVgc_length = 4;
parameter real
  I_gc[1:IVgc_length] = '{4{0.00}};
  V_gc[1:IVgc_length] = '{-5.00, -1.00, 5.00, 10.00};
```

Parameter arrays are subject to the following restrictions:

- You must specify the type of a parameter array in the declaration.
- An array assigned to an instance of a module must be of the exact size of the array bounds of that instance.
- If you change the array size using a parameter assignment, the parameter array must be assigned an array of the new size from the same module as the parameter assignment that changed the parameter array size.

## String Parameters

Use the `string` parameter declaration to declare a parameter of type string.

```
string_parameter_declaration ::=
  parameter string stringparam = constant_expression ;
```

*stringparam* is the name of the string parameter being declared.

*constant\_expression* is the value to be assumed by *stringparam*.

For example, the following code declares a string parameter named `tmdata` and gives it the value `table1.dat`.

```
parameter string tmdata = "table1.dat" ;
```

You can use this parameter to specify the data file for the `$table_model` function as follows:

```
analog begin
  I(d, s) <+ $table_model (V(g, s), V(d, s), tmdata, "I,3CL,3CL");
end
```

## Parameter Aliases

Use the `aliasparam` declaration to define one or more aliases for a parameter. With this capability, you can define alternative names that can be used for overriding module parameter values.

```
aliasparam declaration ::=  
    aliasparam alias_identifier = parameter_identifier ;
```

Parameter aliases are subject to the following restrictions.

- The *alias\_identifier* must not be used for any other object in the module. Equations in the module must reference *parameter\_identifier*, not *alias\_identifier*.
- You must not use both an *alias\_identifier* and its corresponding *parameter\_identifier* to specify a parameter override. Similarly, you must not use multiple aliases corresponding to a single *parameter\_identifier* to specify a parameter override.

For example, the module `nmos` includes the following declarations.

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

The first two instantiations of the module below are valid, but the third is not.

```
nmos #(.trise(5)) m1(.d(d), .g(g), .s(s), .b(b)) ;  
nmos #(.dtemp(5)) m2(.d(), .g(g), .s(s), .b(b)) ;  
nmos #(.trise(5), .dtemp(5)) m3(.d(d), .g(g), .s(s), .b(b)) ; // Illegal.
```

The third instantiation is illegal because overrides are specified for both the parameter `dtemp` and its alias, `trise`.

## Paramsets

Use the `paramset` declaration to declare a set of parameters for a particular module, such that each instance of the paramset need only provide overrides for a smaller number of parameters. The paramset must not contain behavioral code; all of the behavior is determined by the associated module. For information on instantiating paramsets, see [“Overriding Parameter Values by Using Paramsets”](#) on page 223.

```
paramset_declaration ::=  
    {attribute_instance} paramset paramset_name module_or_paramset ;  
    paramset_item_declaration {paramset_item_declaration}  
    paramset_statement { paramset_statement }  
    endparamset  
  
paramset_item_declaration ::=  
    {attribute_instance} parameter_declaration
```



## Cadence Verilog-A Language Reference

### Data Types and Objects

---

```
| {attribute_instance} local_parameter_declaration
| {attribute_instance} string_parameter_declaration
| {attribute_instance} local_string_parameter_declaration
| aliasparam_declaration
| {attribute_instance} integer_declaration
| {attribute_instance} real_declaration

paramset_statement ::=
    .module_parameter_id = constant_expression ;
    | statement
```

*attribute\_instance* is a description attribute, to be used by the simulator when generating help messages for the paramset.

*paramset\_name* is the name of the paramset being defined. Multiple paramsets can be declared using the same *paramset\_name*, but paramsets of the same name must all reference the same module.

*module\_or\_paramset* is the name of a non-structural module with which the paramset is associated or the name of a second paramset. A chain of paramsets can be defined, but the last paramset in the chain must reference a non-structural module.

*module\_parameter\_id* is a parameter of the associated module.

*constant\_expression* is a value to be assigned to the parameter of the associated module. The *constant\_expression* can include numbers, and parameters, but hierarchical out-of-module references to parameters of different modules are unsupported and cannot be included.

*paramset\_statement* can use any statements available for conditional execution but must not include the following:

- Access functions
- Contribution statements
- Event control statements
- Named blocks

Paramset statements can assign values to variables declared in the paramset and the values for such variables do not need to be constant expressions. However, these variables cannot be used to assign values to the parameters of the modules.

Paramsets are subject to the following restrictions:

- Using the `alter` and `altergroup` statements is unsupported when paramsets are used.

- Paramsets cannot be stored in the Cadence library.cell:view configurations, which are sometimes referred to as 5.X configurations.

## Paramset Output Variables

Integer or real variables that are declared with descriptions in the paramset are considered paramset output variables for instances that use the paramset. The following rules apply to paramset output variables and to the output variables of modules referenced by a paramset:

- If a paramset output variable has the same name as a module output variable, the value of the paramset output variable is the value that is reported for any instance that uses the paramset.
- If a paramset variable without a description has the same name as a module output variable, the module output variable of that name is not available for any instance that uses the paramset.

## Genvars

Use the `genvar` declaration to specify a list of integer-valued variables used to compose static expressions for use with behavioral loops.

```
genvar_declaration ::=  
    genvar genvar_identifier {, genvar_identifier}
```

Genvar variables can be assigned only in limited contexts, such as accessing analog signals within behavioral looping constructs. For example, in the following fragment, the genvar variable `i` can only be assigned within the control of the `for` loop. Assignments to the genvar variable `i` can consist of only expressions of static values, such as parameters, literals, and other genvar variables.

```
genvar i ;  
analog begin  
    ...  
    for (i = 0; i < 8; i = i + 1) begin  
        V(out[i]) <+ transition(value[i], td, tr) ;  
    end  
    ...  
end
```

The next example illustrates how genvar variables can be nested.

```
module gen_case(in,out);  
input [0:1] in;  
output [0:1] out;  
electrical [0:1] in;  
electrical [0:1] out;  
genvar i, j;
```

## Cadence Verilog-A Language Reference

### Data Types and Objects

---

```
analog begin
  for( i=1 ; i<0 || i <= 4; i = i + 1 ) begin
    for( j = 0 ; j < 4 ; j = j + 1 ) begin
      $strobe("%d %d", j, i);
    end
  end
  for( j = 0; j < 2; j = j + 1 ) begin
    V(out[j], in[j]) <+ I(out[j], in[j]);
  end
end
endmodule
```

The following example shows how parameters can be assigned to the genvar variables:

```
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
  end
endmodule
```

A *genvar expression* is an expression that consists of only literals and genvar variables. You can also use the `param_given` function in genvar expressions.

## Natures

Use the nature declaration to define a collection of attributes as a nature. The attributes of a nature characterize the analog quantities that are solved for during a simulation. Attributes define the units (such as meter, gram, and newton), access symbols and tolerances associated with an analog quantity, and can define other characteristics as well. After you define a nature, you can use it as part of the definition of disciplines and other natures.

```
nature_declaration ::=
    nature nature_name [ ; ]
    [ nature_descriptions ]
    endnature

nature_name ::=
    nature_identifier

nature_descriptions ::=
    nature_description
    | nature_description nature_descriptions

nature_description ::=
    attribute = constant_expression ;

attribute ::=
    abstol
    | access
    | ddt_nature
    | idt_nature
    | units
    | identifier
    | Cadence_specific_attribute

Cadence_specific_attribute ::=
    huge
    | blowup
    | maxdelta
```

Each of your nature declarations must

- Be named with a unique identifier
- Include all the required attributes listed in [Table 4-3](#) on page 70.
- Be declared at the top level

This requirement means that you cannot nest nature declarations inside other nature, discipline, or module declarations.

The Verilog-A language specification allows you to define a nature in two ways. One way is to define the nature directly by describing its attributes. A nature defined in this way is a *base nature*, one that is not derived from another already declared nature or discipline.

The other way you can define a nature is to derive it from another nature or a discipline. In this case, the new nature is called a *derived nature*.

**Note:** This release of Verilog-A does not support derived natures.

## Declaring a Base Nature

To declare a base nature, you define the attributes of the nature. For example, the following code declares the nature `current` by specifying five attributes. As required by the syntax, the expression associated with each attribute must be a constant expression.

```
nature Mycurrent
  units = "A" ;
  access = I ;
  idt_nature = charge ;
  abstol = 1e-12 ;
  huge = 1e6 ;
endnature
```

Verilog-A provides the predefined attributes described in the “Predefined Attributes” table. Cadence provides the additional attributes described in [Table 4-2](#) on page 70. You can also declare user-defined attributes by declaring them just as you declare the predefined attributes. The Spectre<sup>®</sup> circuit simulator ignores user-defined attributes, but other simulators might recognize them. When you code user-defined attributes, be certain that the name of each attribute is unique in the nature you are defining.

The following table describes the predefined attributes.

**Table 4-1 Predefined Attributes**

---

Attribute	Description
<code>abstol</code>	Specifies a tolerance measure used by the simulator to determine when potential or flow calculations have converged. <code>abstol</code> specifies the maximum negligible value for signals associated with the nature. For more information, see <a href="#">“Convergence”</a> on page 325.
<code>access</code>	Identifies the name of the access function for this nature. When this nature is bound to a potential value, <code>access</code> is the access function for the potential. Similarly, when this nature is bound to a flow value, <code>access</code> is the access function for the flow. Each access function must have a unique name.
<code>units</code>	Specifies the units to be used for the value accessed by the access function.
<code>idt_nature</code>	Specifies a nature to apply when the <code>idt</code> or <code>idtmod</code> operators are used. <b>Note:</b> This release of Verilog-A ignores this attribute.
<code>ddt_nature</code>	Specifies a nature to apply when the <code>ddt</code> operator is used. <b>Note:</b> This release of Verilog-A ignores this attribute.

---

## Cadence Verilog-A Language Reference

### Data Types and Objects

---

The next table describes the Cadence-specific attributes.

**Table 4-2 Cadence-Specific Attributes**

Attribute	Description
huge	Specifies the maximum change in signal value allowed during a single iteration. The simulator uses <code>huge</code> to facilitate convergence when signal values are very large. Default: 45.036e06
blowup	Specifies the maximum allowed value for signals associated with the nature. If the signal exceeds this value, the simulator reports an error and stops running. Default: 1.0e09
maxdelta	Specifies the maximum change allowed on a Newton-Raphson iteration. Default: 0.3

---

The next table specifies the requirements for the predefined and Cadence-specific attributes.

**Table 4-3 Attribute Requirements**

Attribute	Required or optional?	The constant expression must be
abstol	Required	A real value
access	Required for all base natures	An identifier
units	Required for all base natures	A string
idt_nature	Optional	The name of a nature defined elsewhere
ddt_nature	Optional	The name of a nature defined elsewhere
huge	Optional	A real value
blowup	Optional	A real value
maxdelta	Optional	A real value

---

Consider the following code fragment, which declares two base natures.

```
nature Charge
  abstol = 1e-14 ;
  access = Q ;
  units = "coul" ;
  blowup = 1e8 ;
endnature
```

```
nature Current
  abstol = 1e-12 ;
  access = I ;
  units = "A" ;
endnature
```

Both nature declarations specify all the required attributes: `abstol`, `access`, and `units`. In each case, `abstol` is assigned a real value, `access` is assigned an identifier, and `units` is assigned a string.

The `Charge` declaration includes an optional Cadence-specific attribute called `blowup` that ends the simulation if the charge exceeds the specified value.

## Disciplines

Use the `discipline` declaration to specify the characteristics of a discipline. You can then use the discipline to declare nets.

```
discipline_declaration ::=
    discipline discipline_identifier [ ; ]
    [ discipline_description { discipline_description } ]
    enddiscipline

discipline_description ::=
    nature_binding
  | domain_binding

nature_binding ::=
    potential nature_identifier ;
  | flow nature_identifier ;

domain_binding ::=
    domain continuous ;
  | domain discrete ;
```

You must declare a discipline at the top level. In other words, you cannot nest a discipline declaration inside other discipline, nature, or module declarations. Discipline identifiers have global scope, so you can use discipline identifiers to associate nets with disciplines (declare nets) inside any module.

Although you can declare discrete disciplines, you must not instantiate any objects that use such disciplines.

## Binding Natures with Potential and Flow

The disciplines that you declare can bind

- One nature with potential
- One nature with potential and a different nature with flow

- Nothing with either potential or flow

A declaration of this latter form defines an *empty discipline*.

The following examples illustrate each of these forms.

The first example defines a single binding, one between potential and the nature `Voltage`. A discipline with a single binding is called a *signal-flow discipline*.

```
discipline voltage
    potential Voltage ; // A signal-flow discipline must be bound to potential.
enddiscipline
```

The next declaration, for the `electrical` discipline, defines two bindings. Such a declaration is called a *conservative discipline*.

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

When you define a conservative discipline, you must be sure that the nature bound to potential is different from the nature bound to flow.

The third declaration defines an empty discipline. If you do not explicitly specify a domain for an empty discipline, the domain is determined by the connectivity of the net.

```
discipline neutral
enddiscipline

discipline interconnect
    domain continuous
enddiscipline
```

### **Important**

In addition to declaring empty disciplines, you can also use a Verilog-A predefined empty discipline called `wire`.

Use an empty discipline when you want to let the components connected to a net determine which potential and flow natures are used for the net.

Verilog-A supports only the continuous discipline. You can declare a signal as discrete but you cannot otherwise use such a signal.

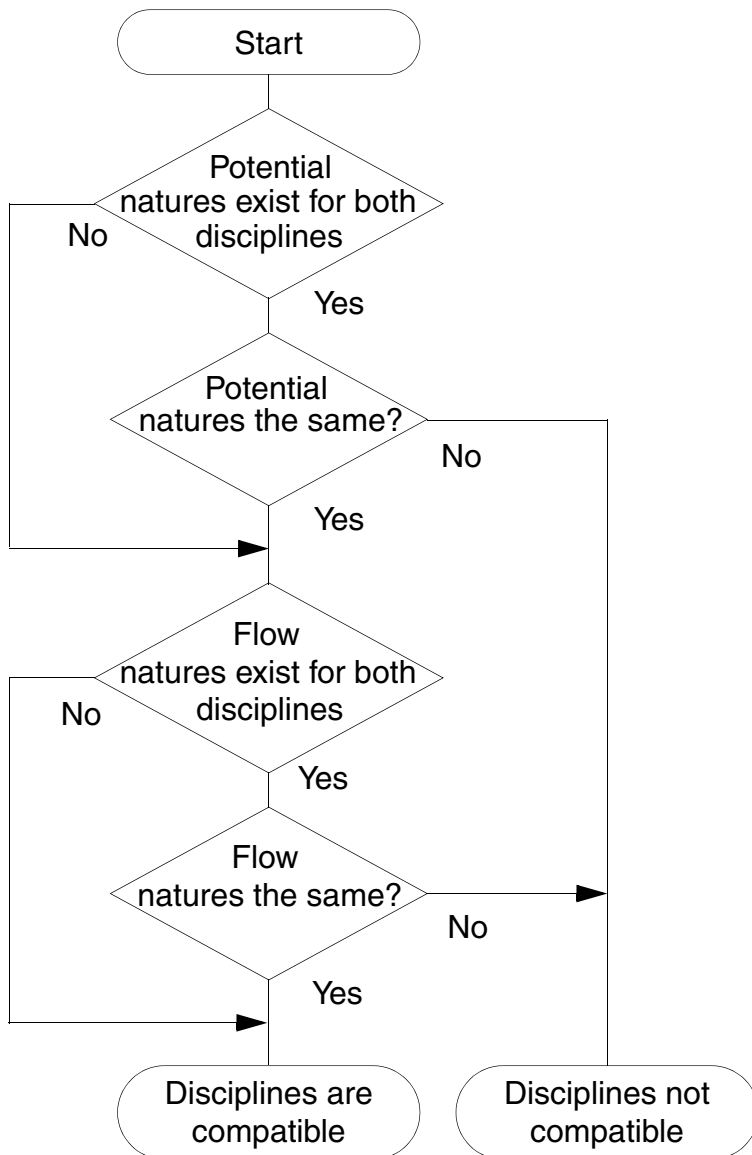
## **Compatibility of Disciplines**

Certain operations in Verilog-A, such as declaring branches, are allowed only if the disciplines involved are compatible. Apply the following rules to determine whether any two disciplines are compatible.



- Any discipline is compatible with itself.
- An empty discipline is compatible with all disciplines.
- Other kinds of continuous disciplines are compatible or not compatible, as determined by following paths through the following figure.

**Figure 4-1 Analog Discipline Compatibility**



## Cadence Verilog-A Language Reference

### Data Types and Objects

---

Consider the following declarations.

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

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

discipline emptydis
enddiscipline

discipline electrical
  potential Voltage ;
  flow Current ;
enddiscipline

discipline sig_flow_v
  potential Voltage ;
enddiscipline
```

To determine whether the `electrical` and `sig_flow_v` disciplines are compatible, follow through the discipline compatibility chart:

1. Both `electrical` and `sig_flow_v` have defined natures for potential. Take the *Yes* branch.
2. In fact, `electrical` and `sig_flow_v` have the same nature for potential. Take the *Yes* branch.
3. `electrical` has a defined nature for flow, but `sig_flow_v` does not. Take the *No* branch to the *Disciplines are compatible* end point.

Now add these declarations to the previous lists.

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

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

discipline mechanical
  potential Position ;
  flow force ;
enddiscipline
```

The `electrical` and `mechanical` disciplines are not compatible.

1. Both disciplines have defined natures for potential. Take the `Yes` branch.
2. The `Position` nature is not the same as the `Voltage` nature. Take the `No` branch to the *Disciplines not compatible* end point.

## Net Disciplines

Use the net discipline declaration to associate nets with previously defined disciplines.

```
net_discipline_declaration ::=
    discipline_identifier [range] list_of_nets ;
    | wire [range] list_of_nets ;
range ::=
    [ msb_expr : lsb_expr ]
list_of_nets ::=
    net_type
    | net_type , list_of_nets
msb_expr ::=
    constant_expr
lsb_expression ::=
    constant_expr
net_type ::=
    net_identifier [range] [= constant_expr | constant_array_expr]
```

You can use the `desc` attribute to specify a description for a net discipline declaration as follows:

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

However, Cadence software does nothing with the information at this time.

The initializers specified with the equals sign in the `net_type` can be used only when the `discipline_identifier` is a continuous discipline. The solver uses the initializer, if provided, as a nodeset value for the potential of the net. A null value in the `constant_array_expr` means that no nodeset value is being specified for that element of the bus. The initializers cannot include out-of-module references.

A net declared without a range is called a *scalar net*. A net declared with a range is called a *vector net*. In this release of Verilog-A, you cannot use parameters to define range limits.

```
magnetic inductor1, inductor2 ; //Declares two scalar nets
electrical [1:10] node1 ; //Declares a vector net
wire [3:0] connect1, connect2 ; //Declares two vector nets
electrical [0:4] bus = {2.3,4.5,,6.0} ; //Declares vector net with nodeset values
```

The following example is illegal because a range, if defined, must be the first item after the discipline identifier and then applies to all of the listed net identifiers.

## Cadence Verilog-A Language Reference

### Data Types and Objects

---

```
electrical AVDD, AVSS, BGAVSS, PD, SUB, [6:1] TRIM ; // Illegal
```

**Note:** Cadence recommends that you specify the direction of a port before you specify the discipline. For example, in the following example the directions for `out` and `in` are specified before the `electrical` discipline declaration.

Consider the following declarations.

```
discipline emptydis
enddiscipline

module comp1 (out, in, unknown1, unknown2) ;
output out ;
input in ;
electrical out, in ;
emptydis unknown1 ; // Declared with an empty discipline
analog
    V(out) <+ 2 * V(in)
endmodule
```

Module `comp1` has four ports: `out`, `in`, `unknown1`, and `unknown2`. The module declares `out` and `in` as `electrical` ports and uses them in the `analog` block. The port `unknown1` is declared with an `empty` discipline and cannot be used in the `analog` block because there is no way to access its signals. However, `unknown1` can be used in the list of ports, where it inherits natures from the ports of module instances that connect to it.

Because `unknown2` appears in the list of ports without being declared in the body of the module, Verilog-A implicitly declares `unknown2` as a scalar port with the default discipline. The default discipline type is `wire`.

Now consider a different example.

```
module five_inputs ( portbus );
input [0:5] portbus;
electrical [0:5] portbus;
real x;
analog begin
    generate i ( 0,4 )
        V(portbus[i]) <+ 0.0;
    end
endmodule
```

The `five_inputs` module uses a port bus. Only one port name, `portbus`, appears in the list of ports but inside the module `portbus` is defined with a range.

Modules `comp1` and `five_inputs` illustrate the two ways you can use nets in a module.

- You can define the ports of a module by giving a list of nets on the module statement.
- You can describe the behavior of a module by declaring and using nets within the body of the module construct.

As you might expect, if you want to describe a conservative system, you must use conservative disciplines to define nets. If you want to describe a signal-flow or mixed signal-flow and conservative system, you can define nets with signal-flow disciplines.

As a result of port connections of analog nets, a single node can be bound to a number of nets of different disciplines.

Current contributions to a node that is bound only to disciplines that have only potential natures, are illegal. The potential of such a node is the sum of all potential contributions, but flow for such a node is not defined.

Nets of signal flow disciplines in modules must not be bound to inout ports and you must not contribute potential to input ports.

To access the `abstol` associated with a nets's potential or flow natures, use the form

```
net.potential.abstol
```

or

```
net.flow.abstol
```

For an example, see [“Cross Event”](#) on page 120.

## Named Branches

Use the branch declaration to declare a path between two nets of continuous discipline. Cadence recommends that you use named branches, especially when debugging with Tcl commands because, for example, it is easier to type `value branch1` than it is to type `value \vect1[5] vect2[1]` and then compute the difference between the returned value.

```
branch_declaration ::=
    branch list_of_branches ;
list_of_branches ::=
    terminals list_of_branch_identifiers
terminals ::=
    ( scalar_net_identifier )
    | ( scalar_net_identifier , scalar_net_identifier )
list_of_branch_identifiers ::=
    branch_identifier
    | branch_identifier , list_of_branch_identifiers
```

*scalar\_net\_identifier* must be either a scalar net or a single element of a vector net.

You can declare branches only in a module. You must not combine explicit and implicit branch declarations for a single branch. For more information, see [“Implicit Branches”](#) on page 78.

## Cadence Verilog-A Language Reference

### Data Types and Objects

---

The scalar nets that the branch declaration associates with a branch are called the *branch terminals*. If you specify only one net, Verilog-A assumes that the other is ground. The branch terminals must have compatible disciplines. For more information, see [“Compatibility of Disciplines”](#) on page 72.

Consider the following declarations.

```
voltage [5:0] vec1 ;           // Declares a vector net
voltage [1:6] vec2 ;           // Declares a vector net
voltage sca1 ;                 // Declares a scalar net
voltage sca2 ;                 // Declares a scalar net
branch (vec1[5],vec2[1]) branch1, (sca1,sca2) branch2 ;
```

branch1 is legally declared because each branch terminal is a single element of a vector net. The second branch, branch2, is also legally declared because nodes sca1 and sca2 are both scalar nets.

## Implicit Branches

As Cadence recommends, you can refer to a named branch with only a single identifier. Alternatively, you might find it more convenient or clearer to refer to branches by their branch terminals. Most of the examples in this reference, including the following example, use this form of implicit branch declaration. You must not, however, combine named and implicit branch declarations for a single branch.

```
module diode (a, c) ;
inout a, c ;
electrical a, c ;
parameter real rs=0, is=1e-14, tf=0, cjo=0, phi=0.7 ;
parameter real kf=0, af=1, ef=1 ;
analog begin
    I(a, c) <+ is*(limexp((V(a, c)-rs*I(a, a))/\$vt) - 1);
    I(a, c) <+ white_noise(2* `P_Q * I(a, c)) ;
    I(a, c) <+ flicker_noise(kf*pow(abs(I(a, c)),af),ef);
end
endmodule
```

The previous example using implicit branches is equivalent to the following example using named branches.

```
module diode (a, c) ;
inout a, c ;
electrical a, c ;
branch (a,c) diode ; // Declare named branch
parameter real rs=0, is=1e-14, tf=0, cjo=0, phi=0.7 ;
parameter real kf=0, af=1, ef=1 ;
analog begin
    I(diode) <+ is*(limexp((V(diode)-rs*I(<a>))/\$vt) - 1);
    I(diode) <+ white_noise(2* `P_Q * I(diode)) ;
    I(diode) <+ flicker_noise(kf*pow(abs(I(diode)),af),ef);
end
endmodule
```

## Output Variables

You can register a variable as an output variable as specified in section 3.2.1 of the Verilog-AMS LRM Version 2.3.1. If you do not register any variables as output variables, the program considers all variables to be output variables. Using the Spectre circuit simulator, you can save output variables using the `save` statement and you can view operating point values for them using `info` analysis.

# Cadence Verilog-A Language Reference

## Data Types and Objects

---



---

## Statements for the Analog Block

---

This chapter describes the assignment statements and the procedural control constructs and statements that the Cadence<sup>®</sup> Verilog<sup>®</sup>-A language supports within the analog block. For information, see the indicated locations. The constructs and statements discussed include

- [Analog Initial Block](#) on page 82
- [Procedural Assignment Statements in the Analog Block](#) on page 82
- [Branch Contribution Statement](#) on page 83
- [Indirect Branch Assignment Statement](#) on page 85
- [Sequential Block Statement](#) on page 86
- [Conditional Statement](#) on page 86
- [Case Statement](#) on page 87
- Loop statements, including
  - [Repeat Statement](#) on page 88
  - [While Statement](#) on page 89
  - [For Statement](#) on page 89
- [Generate Statement](#) on page 90

## Analog Initial Block

The analog initial block is supported for simulation initialization purposes. The block (which is a special analog procedural block), begins with the keywords `analog initial`. 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 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 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.

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). 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 is re-executed so that the new value is taken into account.

## Assignment Statements

There are several kinds of assignment statements in Verilog-A: the procedural assignment statement, the branch contribution statement, and the indirect branch assignment statement. You use the procedural assignment statement to modify integer and real variables and you use the branch contribution and indirect branch assignment statements to modify branch values such as potential and flow.

### Procedural Assignment Statements in the Analog Block

Use the procedural assignment statement to modify integer and real variables.

## Cadence Verilog-A Language Reference

### Statements for the Analog Block

---

```
procedural_assignment ::=
    lexpr = expression ;

lexpr ::=
    integer_identifier
    | real_identifier
    | array_element

array_element ::=
    integer_identifier [ constant_expression ]
    | real_identifier [ constant_expression ]
```

The left-hand operand of the procedural assignment must be a modifiable integer or real variable or an element of an integer or real array. The type of the left-hand operand determines the type of the assignment.

The right-hand operand can be any arbitrary scalar expression constituted from legal operands and operators.

In the following code fragment, the variable `phase` is assigned a real value. The value must be real because `phase` is defined as a real variable.

```
real phase ;
analog begin
    phase = idt( gain*V(in) ) ;
```

You can also use procedural assignment statements to modify array values. For example, if `r` is declared as

```
real r[0:3], sum ;
```

you can make assignments such as

```
r[0] = 10.1 ;
r[1] = 11.1 ;
r[2] = 12.1 ;
r[3] = 13.1 ;
sum = r[0] + r[1] + r[2] + r[3] ;
```

## Branch Contribution Statement

Use the branch contribution statement to modify signal values.

```
branch_contribution ::=
    bvalue <+ expression ;

bvalue ::=
    access_identifier ( analog_signal_list )

analog_signal_list ::=
    branch_identifier
    | node_or_port_identifier
    | node_or_port_identifier , node_or_port_identifier
```

`bvalue` specifies a source branch signal. `bvalue` must consist of an access function applied to a branch. `expression` can be linear, nonlinear, or dynamic.

## Cadence Verilog-A Language Reference

### Statements for the Analog Block

---

Branch contribution statements must be placed within the analog block.

As discussed in the following list, the branch contribution statement differs in important ways from the procedural assignment statement.

- You can use the procedural assignment statement only for variables, whereas you can use the branch contribution statement only for access functions.
- Using the procedural assignment statement to assign a number to a variable overrides the number previously contained in that variable. Using the branch contribution statement, however, adds to any previous contribution. (Contributions to flow can be viewed as adding new flow sources in parallel with previous flow sources. Contributions to value can be viewed as adding new value sources in series with previous value sources.)

### Evaluation of a Branch Contribution Statement

For source branch contributions, the simulator evaluates the branch contribution statement as follows:

1. The simulator evaluates the right-hand operand.
2. The simulator adds the value of the right-hand operand to any previously retained value for the branch.
3. At the end of the evaluation of the analog block, the simulator assigns the summed value to the source branch.

For example, given a pair of nodes declared with the `electrical` discipline, the code fragment

```
V(n1, n2) <+ expr1 ;  
V(n1, n2) <+ expr2 ;
```

is equivalent to

```
V(n1, n2) <+ expr1 + expr2 ;
```

### Creating a Switch Branch



When you contribute a flow to a branch that already has a value retained for potential, the simulator discards the value for potential and converts the branch to a flow source. Conversely, when you contribute a potential to a branch that already has a value retained for flow, the simulator discards the value for flow and converts

the branch to a potential source. Branches converted from flow sources to potential sources, and vice versa, are known as *switch branches*. For additional information, see [“Switch Branches”](#) on page 331.

## Indirect Branch Assignment Statement

Use the indirect branch assignment statement when it is difficult to separate the target from the equation.

```
indirect_branch_assignment ::=
    Target : equation ;

target ::=
    bvalue

equation ::=
    nexpr == expression

nexpr ::=
    bvalue
    | ddt ( bvalue )
    | idt ( bvalue )
    | idtmod ( bvalue )
```

An indirect branch assignment has this format:

```
V(out) : V(in) == 0 ;
```

Read this as “find  $V(\text{out})$  such that  $V(\text{in})$  is zero.” This example says that `out` should be driven with a voltage source and the voltage should be such that the given equation is satisfied. Any branches referenced in the equation are only probed and not driven, so in this example,  $V(\text{in})$  acts as a voltage probe.

Indirect branch assignments can be used only within the analog block.

The next example models an ideal operational amplifier with infinite gain. The indirect assignment statement says “find  $V(\text{out})$  such that  $V(\text{pin}, \text{nin})$  is zero.”

```
module opamp (out, pin, nin) ;
output out ;
input pin, nin ;
voltage out, pin, nin ;
analog
    V(out) : V(pin, nin) == 0 ; // Indirect assignment
endmodule
```

Indirect assignments are incompatible with assignments made with the branch contribution statement. If you indirectly assign a value to a branch, you cannot then contribute to the branch by using the branch contribution statement.

## Sequential Block Statement

Use a sequential block when you want to group two or more statements together so that they act like a single statement.

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

For information on `statement`, see [“Defining Module Analog Behavior”](#) on page 39.

The statements included in a sequential block run sequentially.

If you add a block identifier, you can also declare local variables for use within the block. All the local variables you declare are static. In other words, a unique location exists for each local variable, and entering or leaving the block does not affect the value of a local variable.

The following code fragment uses two named blocks, declaring a local variable in each of them. Although the variables have the same name, the simulator handles them separately because each variable is local to its own block.

```
integer j ;
...
for ( j = 0 ; j < 10 ; j=j+1 ) begin
    if ( j%2 ) begin : odd
        integer j ; // Declares a local variable
        j = j+1 ;
        $display ("Odd numbers counted so far = %d" , j ) ;
    end else begin : even
        integer j ; // Declares a local variable
        j = j+1 ;
        $display ("Even numbers counted so far = %d" , j ) ;
    end
end
```

Each named block defines a new scope. For additional information, see [“Scope Rules”](#) on page 49.

## Conditional Statement

Use the conditional statement to run a statement under the control of specified conditions.

```
conditional_statement ::=
    if ( expression ) statement1
    [ else statement2 ]
```

## Cadence Verilog-A Language Reference

### Statements for the Analog Block

---

If *expression* evaluates to a nonzero number (true), the simulator executes *statement1*. If *expression* evaluates to zero (false) and the `else` statement is present, the simulator skips *statement1* and executes *statement2*.

If *expression* consists entirely of genvar expressions, literal numerical constants, parameters, or the analysis function, *statement1* and *statement2* can include analog operators.

The simulator always matches an `else` statement with the closest previous `if` that lacks an `else`. In the following code fragment, for example, the first `else` goes with the inner `if`, as shown by the indentation.

```
if (index > 0)
    if (i > j) // The next else belongs to this if
        result = i ;
    else // This else belongs to the previous if
        result = j ;
else $strobe ("Index < 0"); // This else belongs to the first if
```

The following code fragment illustrates a particularly useful form of the `if-else` construct.

```
if ((value > 0)&&(value <= 1)) $strobe("Category A");
else if ((value > 1)&&(value <= 2)) $strobe("Category B");
else if ((value > 2)&&(value <= 3)) $strobe("Category C");
else if ((value > 3)&&(value <= 4)) $strobe("Category D");
else $strobe("Illegal value");
```

The simulator evaluates the expressions in order. If any one of them is true, the simulator runs the associated statement and ends the whole chain. The last `else` statement handles the default case, running if none of the other expressions is true.

## Case Statement

Use the `case` construct to control which one of a series of statements runs.

```
case_statement ::=
    case ( expression ) case_item { case_item } endcase
case_item ::=
    test_expression { , test_expression } : statement
    | default [ : ] statement
```

The default statement is optional. Using more than one default statement in a case construct is illegal.

The simulator evaluates each *test\_expression* in turn and compares it with *expression*. If there is a match, the statement associated with the matching *test\_expression* runs. If none of the expressions in *test\_expression* matches *expression* and if you coded a default `case_item`, the default statement runs. If all

## Cadence Verilog-A Language Reference

### Statements for the Analog Block

---

comparisons fail and you did not code a default `case_item`, none of the associated statements runs.

If *expression* and *text\_expression* are genvar expressions, parameters, or the analysis function, *statement* can include analog operators; otherwise, *statement* cannot include analog operators.

The following code fragment determines what range `value` is in. For example, if `value` is 1.5 the first comparison fails. The second *test\_expression* evaluates to 1 (true), which matches the case expression, so the `$strobe("Category B")` statement runs.

```
real value ;
...
case (1)
  ((value > 0)&&(value <= 1)) : $strobe("Category A");
  ((value > 1)&&(value <= 2)) : $strobe("Category B");
  ((value > 2)&&(value <= 3)) : $strobe("Category C");
  ((value > 3)&&(value <= 4)) : $strobe("Category D");
  value <= 0 , value >= 4 : $strobe("Out of range");
  default $strobe("Error. Should never get here.");
endcase
```

## Repeat Statement

Use the `repeat` statement when you want a statement to run a fixed number of times.

```
repeat_statement ::=
  repeat ( constant_expression ) statement
```

*statement* must not include any analog operators, event control statements, and contribution statements. For additional information on analog operators, see [“Analog Operators”](#) on page 161.

The following example code repeats the loop exactly 10 times while summing the first 10 digits.

```
integer i, total ;
...
i = 0 ;
total = 0 ;
repeat (10) begin
  i = i + 1 ;
  total = total + i ;
end
```



## While Statement

Use the `while` statement when you want to be able to leave a loop when an expression is no longer valid.

```
while_statement ::=
    while ( expression ) statement
```

The `while` loop evaluates *expression* at each entry into the loop. If *expression* is nonzero (true), *statement* runs. If *expression* starts out as zero (false), *statement* never runs.

*statement* must not include any analog operators, event control statements, and contribution statements. For additional information on analog operators, see [“Analog Operators”](#) on page 161.

The following code fragment counts the number of random numbers generated before `rand` becomes zero.

```
integer rand, count ;
...
    rand = abs($random % 10) ;
    count = 0 ;
    while (rand) begin
        count = count + 1 ;
        rand = abs($random % 10) ;
    end ;
    $strobe ("Count is %d", count) ;
```

## For Statement

Use the `for` statement when you want a statement to run a fixed number of times.

```
for_statement ::=
    for ( initial_assignment ; expression ;
        step_assignment ) statement
```

If *initial\_assignment*, *expression*, and *step\_assignment* are genvar expressions, the statement can include analog operators; otherwise, the *statement* must not include any analog operators. In addition, the statement must not include event control statements and contribution statements. For additional information on analog operators, see [“Analog Operators”](#) on page 161.

Use *initial\_assignment* to initialize an integer loop control variable that controls the number of times the loop executes. The simulator evaluates *expression* at each entry into the loop. If *expression* evaluates to zero, the loop terminates. If *expression* evaluates to a nonzero value, the simulator first runs *statement* and then runs

*step\_assignment*. *step\_assignment* is usually defined so that it modifies the loop control variable before the simulator evaluates *expression* again.

For example, to sum the first 10 even numbers, the `repeat` loop given earlier could be rewritten as a `for` loop.

```
integer j, total ;
...
    total = 0 ;
    for ( j = 2; j < 22; j = j + 2 )
        total = total + j ;
```

## Generate Statement

**Note:** The `generate` statement is obsolete. To comply with current practice, use the `genvar` statement instead.

The `generate` statement is a looping construct that is unrolled at compile time. Use the `generate` statement to simplify your code or when you have a looping construct that contains analog operators. The `generate` statement can be used only within the analog block. The `generate` statement is supported only for backward compatibility.

```
generate_statement ::=
    generate index_identifier ( start_expr ,
        end_expr [ , incr_expr ] ) statement
start_expr ::=
    constant_expression
end_expr ::=
    constant_expression
incr_expr ::=
    constant_expression
```

*index\_identifier* is an identifier used in *statement*. When *statement* is unrolled, each occurrence of *index\_identifier* found in *statement* is replaced by a constant. You must be certain that nothing inside *statement* modifies the index.

In the first unrolled instance of *statement*, the compiler replaces each occurrence of *index\_identifier* by the value *start\_expr*. In the second instance, the compiler replaces each *index\_identifier* by the value *start\_expr* plus *incr\_expr*. In the third instance, the compiler replaces each *index\_identifier* by the value *start\_expr* plus twice the *incr\_expr*. This process continues until the replacement value is greater than the value of *end\_expr*.

If you do not specify *incr\_expr*, it takes the value +1 if *end\_expr* is greater than *start\_expr*. If *end\_expr* is less than *start\_expr*, *incr\_expr* takes the value -1 by default.

## Cadence Verilog-A Language Reference

### Statements for the Analog Block

---

The values of the `start_expr`, `end_expr`, and `incr_expr` determine how the `generate` statement behaves.

---

If	And	Then the generate statement
<code>start_expr &gt; end_expr</code>	<code>incr_expr &gt; 0</code>	does not execute
<code>start_expr &lt; end_expr</code>	<code>incr_expr &lt; 0</code>	does not execute
<code>start_expr = end_expr</code>		executes once

---

As an example of using the `generate` statement, consider the following module, which implements an analog-to-digital converter.

```
`define BITS 4
module adc (in, out) ;
input in ;
output [0: `BITS - 1] out ;
electrical in ;
electrical [0: `BITS - 1] out ;
parameter fullscale = 1.0, tdelay = 0.0, trantime = 10n ;
real samp, half ;

analog begin
    half = fullscale/2.0 ;
    samp = V(in) ;
    generate i ( `BITS - 1,0) begin // default increment = -1
        V(out[i]) <+ transition(samp > half, tdelay, trantime);
        if (samp > half) samp = samp - half ;
        samp = 2.0 * samp ;
    end
end
endmodule
```

Module `adc` is equivalent to the following module coded without using the `generate` statement.

```
`define BITS 4
module adc_unrolled (in, out) ;
input in ;
output [0: `BITS - 1] out ;
electrical in;
electrical [0: `BITS - 1] out ;
parameter fullscale = 1.0, tdelay = 0.0, trantime = 10n ;
real samp, half ;

analog begin
    half = fullscale/2.0 ;
    samp = V(in) ;
    V(out[3]) <+ transition(samp > half, tdelay, trantime);
    if (samp > half) samp = samp - half ;
    samp = 2.0 * samp ;
    V(out[2]) <+ transition(samp > half, tdelay, trantime);
    if (samp > half) samp = samp - half ;
```

## Cadence Verilog-A Language Reference

### Statements for the Analog Block

---

```
samp = 2.0 * samp ;
V(out[1]) <+ transition(samp > half, tdelay, trantime);
if (samp > half) samp = samp - half ;
samp = 2.0 * samp ;
V(out[0]) <+ transition(samp > half, tdelay, trantime);
if (samp > half) samp = samp - half ;
samp = 2.0 * samp ;
end
endmodule
```

**Note:** Because the `generate` statement is unrolled at compile time, you cannot use the Verilog-A debugging utility to examine the value of `index_identifier` or to evaluate expressions that contain `index_identifier`. For example, if `index_identifier` is `i`, you cannot use a debugging command like `print i` nor can you use a command like `print{a[i]}`.

---

## Operators for Analog Blocks

---

This chapter describes the operators that you can use in analog blocks and explains how to use them to form expressions. For basic definitions, see

- [Unary Operators](#) on page 95
- [Binary Operators](#) on page 96
- [Bitwise Operators](#) on page 99
- [Ternary Operator](#) on page 100

For information about precedence and short-circuiting, see

- [Operator Precedence](#) on page 101
- [Expression Short-Circuiting](#) on page 101

For information about string operators and functions, see

- [String Operators and Functions](#) on page 101

## Overview of Operators

An *expression* is a construct that combines operands with operators to produce a result that is a function of the values of the operands and the semantic meaning of the operators. Any legal operand is also an expression. You can use an expression anywhere Verilog-A requires a value.

A *constant expression* is an expression whose operands are constant numbers and previously defined parameters and whose operators all come from among the unary, binary, and ternary operators described in this chapter.

The operators listed below, with the single exception of the conditional operator, associate from left to right. That means that when operators have the same precedence, the one farthest to the left is evaluated first. In this example

`A + B - C`

the simulator does the addition before it does the subtraction.

When operators have different precedence, the operator with the highest precedence (the smallest precedence number) is evaluated first. In this example

`A + B / C`

the division (which has a precedence of 2) is evaluated before the addition (which has a precedence of 3). For information on precedence, see [“Operator Precedence”](#) on page 101.

You can change the order of evaluation with parentheses. If you code

`(A + B) / C`

the addition is evaluated before the division.

The operators divide into three groups, according to the number of operands the operator requires. The groups are the unary operators, the binary operators, and the ternary operator.

## Unary Operators

The unary operators each require a single operand. The unary operators have the highest precedence of all the operators discussed in this chapter.

### Unary Operators

Operator	Precedence	Definition	Type of Operands Allowed	Example or Further Information
+	1	Unary plus	Integer, real	<code>I = +13; // I = 13</code> <code>I = +(-13); // I = -13</code>
-	1	Unary minus	Integer, real	<code>R = -13.1; // R = -13.1</code> <code>I = -(4-5); // I = 1</code>
!	1	Logical negation	Integer, real	<code>I = !(1==1); // I = 0</code> <code>I = !(1==2); // I = 1</code> <code>I = !13.2; // I = 0</code> <i>/*Result is zero for a non-zero operand*/</i>
~	1	Bitwise unary negation	Integer	See the <a href="#">Bitwise Unary Negation Operator</a> figure on page 100.
&	1	Unary reduction AND	integer	See “ <a href="#">Unary Reduction Operators.</a> ”
~&	1	Unary reduction NAND	integer	See “ <a href="#">Unary Reduction Operators.</a> ”
	1	Unary reduction OR	integer	See “ <a href="#">Unary Reduction Operators.</a> ”
~	1	Unary reduction NOR	integer	See “ <a href="#">Unary Reduction Operators.</a> ”
^	1	Unary reduction exclusive OR	integer	See “ <a href="#">Unary Reduction Operators.</a> ”
^~ or ~^	1	Unary reduction exclusive NOR	integer	See “ <a href="#">Unary Reduction Operators.</a> ”

## Unary Reduction Operators

The unary reduction operators perform bitwise operations on single operands and produce a single bit result. The reduction `AND`, reduction `OR`, and reduction `XOR` operators first apply the following logic tables between the first and second bits of the operand to calculate a result.

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

Then for the second and subsequent steps, these operators apply the same logic table to the previous result and the next bit of the operand, continuing until there is a single bit result.

The reduction `NAND`, reduction `NOR`, and reduction `XNOR` operators are calculated in the same way, except that the result is inverted.

#### Unary Reduction AND Operator

<b>&amp;</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	0
<b>1</b>	0	1

#### Unary Reduction OR Operator

<b> </b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	1

#### Unary Reduction Exclusive OR Operator

<b>^</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	0

## Binary Operators

The binary operators each require two operands.

#### Binary Operators

---

<b>Operator</b>	<b>Precedence</b>	<b>Definition</b>	<b>Type of Operands Allowed</b>	<b>Example or Further Information</b>
<code>+</code>	3	$a$ plus $b$	Integer, real	<code>R = 10.0 + 3.1; // R = 13.1</code>



## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

#### Binary Operators, *continued*

Operator	Precedence	Definition	Type of Operands Allowed	Example or Further Information
-	3	$a$ minus $b$	Integer, real	<code>I = 10 - 13; // I = -3</code>
*	2	$a$ multiplied by $b$	Integer, real	<code>R = 2.2 * 2.0; // R = 4.4</code>
**	1	$a$ raised to the power of $b$	Integer, real	<code>R = 2 ** 3 // R = 8</code>
/	2	$a$ divided by $b$	Integer, real	<code>I = 9 / 4; // I = 2</code> <code>R = 9.0 / 4; // R = 2.25</code>
%	2	$a$ modulo $b$	Integer, real	<code>I = 10 % 5; // I = 0</code> <code>I = -12 % 5; // I = -2</code> <code>R = 10 % 3.75 // R = 2.5</code> <i>/*The result takes sign of the first operand.*/</i>
<	5	$a$ less than $b$ ; evaluates to 0 or 1	Integer, real	<code>I = 5 &lt; 7; // I = 1</code> <code>I = 7 &lt; 5; // I = 0</code>
>	5	$a$ greater than $b$ ; evaluates to 0 or 1	Integer, real	<code>I = 5 &gt; 7; // I = 0</code> <code>I = 7 &gt; 5; // I = 1</code>
<=	5	$a$ less than or equal to $b$ ; evaluates to 0 or 1	Integer, real	<code>I = 5.0 &lt;= 7.5; // I = 1</code> <code>I = 5.0 &lt;= 5.0; // I = 1</code> <code>I = 5 &lt;= 4; // I = 0</code>
>=	5	$a$ greater than or equal to $b$ ; evaluates to 0 or 1	Integer, real	<code>I = 5.0 &gt;= 7; // I = 0</code> <code>I = 5.0 &gt;= 5; // I = 1</code> <code>I = 5.0 &gt;= 4.8; // I = 1</code>
==	6	$a$ equal to $b$ ; evaluates to 0, 1, or $x$ (if any bit of $a$ or $b$ is $x$ or $z$ ).	Integer, real	<code>I = 5.2 == 5.2; // I = 1</code> <code>I = 5.2 == 5.0; // I = 0</code> <code>I = 1 == 1'bx; // I = x</code>

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

#### Binary Operators, *continued*

Operator	Precedence	Definition	Type of Operands Allowed	Example or Further Information
!=	6	$a$ not equal to $b$ ; evaluates to 0, 1, or $x$ (if any bit of $a$ or $b$ is $x$ or $z$ ).	Integer, real	<code>I = 5.2 != 5.2; // I = 0</code> <code>I = 5.2 != 5.0; // I = 1</code>
&&	10	Logical AND; evaluates to 0 or 1	Integer, real	<code>I = (1==1) &amp;&amp; (2==2); // I = 1</code> <code>I = (1==2) &amp;&amp; (2==2); // I = 0</code> <code>I = -13 &amp;&amp; 1; // I = 1</code>
	11	Logical OR; evaluates to 0 or 1	Integer, real	<code>I = (1==2)    (2==2); // I = 1</code> <code>I = (1==2)    (2==3); // I = 0</code> <code>I = 13    0; // I = 1</code>
&	7	Bitwise binary AND	Integer	See the <a href="#">Bitwise Binary AND Operator</a> figure on page 99.
	9	Bitwise binary OR	Integer	See the <a href="#">Bitwise Binary OR Operator</a> figure on page 99.
^	8	Bitwise binary exclusive OR	Integer	See the <a href="#">Bitwise Binary Exclusive OR Operator</a> figure on page 99.
^~	8	Bitwise binary exclusive NOR (Same as ~^)	Integer	See the <a href="#">Bitwise Binary Exclusive NOR Operator</a> figure on page 99.
~^	8	Bitwise binary exclusive NOR (Same as ^~)	Integer	See the <a href="#">Bitwise Binary Exclusive NOR Operator</a> figure on page 99.
<<	4	$a$ shifted $b$ bits left	Integer	<code>I = 1 &lt;&lt; 2; // I = 4</code> <code>I = 2 &lt;&lt; 2; // I = 8</code> <code>I = 4 &lt;&lt; 2; // I = 16</code>
>>	4	$a$ shifted $b$ bits right	Integer	<code>I = 4 &gt;&gt; 2; // I = 1</code> <code>I = 2 &gt;&gt; 2; // I = 0</code>
or	11	Event OR	Event expression	<code>@(initial_step or cross(V(vin)-1))</code>

## Bitwise Operators

The bitwise operators evaluate to integer values. Each operator combines a bit in one operand with the corresponding bit in the other operand to calculate a result according to these logic tables.

### Bitwise Binary AND Operator

<b>&amp;</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	0
<b>1</b>	0	1

### Bitwise Binary OR Operator

<b> </b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	1

### Bitwise Binary Exclusive OR Operator

<b>^</b>	<b>0</b>	<b>1</b>
<b>0</b>	0	1
<b>1</b>	1	0

### Bitwise Binary Exclusive NOR Operator

<b>^~ or ~^</b>	<b>0</b>	<b>1</b>
<b>0</b>	1	0
<b>1</b>	0	1

### Bitwise Unary Negation Operator

~		
0		1
1		0

## Ternary Operator

There is only one ternary operator, the conditional operator. The conditional operator has the lowest precedence of all the operators listed in this chapter.

### Conditional Operator

Operator	Precedence	Definition	Type of Operands Allowed	Example or Further Information
?:	12	$exp ? t\_exp : f\_exp$	Valid expressions	I= 2==3 ? 1:0; // I = 0 R= 1==1 ? 1.0:0.0; // R=1.0

A complete conditional operator expression looks like this:

```
conditional_expr ? true_expr : false_expr
```

If *conditional\_expr* is true, the conditional operator evaluates to *true\_expr*, otherwise to *false\_expr*.

The conditional operator is right associative.

This operator performs the same function as the `if-else` construct. For example, the contribution statement

```
V(out) <+ V(in) > 2.5 ? 0.0 : 5.0 ;
```


is equivalent to

```
If (V(in) > 2.5)
    V(out) <+ 0.0 ;
else
    V(out) <+ 5.0 ;
```

## Operator Precedence

The following table summarizes the precedence information for the unary, binary, and ternary operators. Operators at the top of the table have higher precedence than operators lower in the table.

Precedence	Operators	
1	+ - ! ~ (unary)	Highest precedence
2	* / %	
3	+ - (binary)	
4	<< >>	
5	< <= > >=	
6	== !=	
7	&	
8	^ ~^ ^~	
9		
10	&&	
11		
12	?: (conditional operator)	Lowest precedence



## Expression Short-Circuiting

Sometimes the simulator can determine the value of an expression containing logical AND ( && ), logical OR ( || ), or bitwise AND ( & ) without evaluating the entire expression. By taking advantage of such expressions, the simulator operates more efficiently.

## String Operators and Functions

The string operators and functions are for manipulating and comparing strings. The operands can be string parameters provided that the string parameters are not changed. The software supports string operators and functions only in Verilog-A modules that you include in your design using an `ahdl_include` statement.

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

Cadence recommends using the Verilog-A string functions listed in the following table. These functions are adapted from SystemVerilog and though they are non-standard now, they are expected to become part of the Verilog-A standard in the future.

**Table 6-1 Verilog-A String Functions**

Function	Description	Detailed Information
<code>\$sscanf (string_format {, arg})</code>	Reads bytes from a string, interprets the bytes according to the specified <i>string_format</i> format and stores the result in arguments.	<a href="#">“\$sscanf”</a> on page 104.
<code>== != &lt; &gt; &gt;= &lt;=</code>	Compare two strings alphabetically and lexicographically.	<a href="#">“Comparison Operators”</a> on page 103.
<code>des_str = src_str</code>	Copies <i>src_str</i> to <i>des_str</i> .	<a href="#">“String Copy Operator”</a> on page 103.
<code>{str_des, str_src}</code>	Appends (concatenates) <i>src_str</i> to <i>des_str</i> .	<a href="#">“Concatenation Operator”</a> on page 103.
<code>int_as_str.atoi()</code>	Converts a string, <i>int_as_str</i> , to an integer.	<a href="#">“atoi”</a> on page 106.
<code>real_as_str.atoreal()</code>	Converts a string, <i>real_as_str</i> , to a real.	<a href="#">“atoreal”</a> on page 106.
<code>str.getc()</code>	Returns the ASCII code of the first character of <i>string1</i> .	<a href="#">“getc”</a> on page 107.
<code>str.len()</code>	Returns the number of characters in <i>str</i> .	<a href="#">“len”</a> on page 107.
<code>str.substr(start_pos, end_pos)</code>	Returns the substring of <i>str</i> between <i>start_pos</i> and <i>end_pos</i> , inclusive.	<a href="#">“substr”</a> on page 110.

## String Operator Details

This section gives information about the string comparison, copy, and concatenation operators.

### Comparison Operators

Use the string comparison operators to compare two strings alphabetically and lexicographically. The lexicographic order used is that of the ASCII code.

```
comparison_operator ::=
    str1 == str2
  | str1 != str2
  | str1 < str2
  | str1 <= str2
  | str1 > str2
  | str1 >= str2
```

`str1` and `str2` can both be of type string or one of them can be a string literal.

The equality comparison (`==`) returns 1 if the two string are equal and returns 0 otherwise. The inequality comparison (`!=`) returns 1 if the two strings are not equal and returns 0 if they are equal. The other comparison operators return 1 if the condition is true using the lexicographical ordering of the two strings.

For example,

```
inputStr = "YourFriend";
check = (inputStr == "YourFriend" ); // Returns 1
```

### String Copy Operator

Use the string copy operator to copy a string.

```
string_copy_operator ::=
    str2 = str1
```

For example,

```
des_str = src_str;
```

copies `src_str` to `des_str`.

### Concatenation Operator

Use the concatenation operator to append (concatenate) multiple strings together.

```
string_concatenation_operator ::=
    { str1, str2....strm }
```

For example,

```
str_des={str_des, str_src, str_out};
```

appends (concatenates) *str\_src* and *str\_out* to *str\_des*.

## String Replication Operator

Use the string replication operator to append (concatenate) a string a number of times.

```
string_replication_operator ::=  
    { multiplier {str} }
```

For example,

```
str_des={5{"tr"}}
```

appends (concatenates) *tr* five times, as follows:

```
trtrtrtrtr
```

## String Function Details

This section gives information about the string functions.

For functions that refer to positions within the string, note that the first character in a string is considered to be at position 0, the second character in a string is at position 1, and so on.

### **\$sscanf**

Use the `$sscanf` function to read characters, interpret them according to the specified format, and store the results in the specified arguments.

```
$sscanf_function ::=  
    $sscanf(string, format, arg)
```

The behavior of `$sscanf` is undefined in case the arguments specified for the format are insufficient. If there are more arguments than needed for the specified format, the additional arguments are ignored. The least significant bits of the converted input are transferred if the size of the specified argument is too small to accommodate the converted input.

The format, which is a string expression, contains the specifications for conversion. A conversion specification directs the conversion of the next string of non-space characters and stores the result in the variable specified with the corresponding argument unless the `*` character is specified. You can use the following within the string:



## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

- White space characters, which include blanks, tabs, new lines characters, or form feeds. When specified, white spaces are ignored and the input is read up to the next nonwhite space character. Even null characters are considered as white spaces.
- Except %, any character that should match the next character of the input stream.
- Conversion specifications that include:
  - the character %
  - an assignment character \* that skips a string of nonspace characters until it extends to the next inappropriate character or until the specified maximum field width (optional)
  - a decimal digit string that specifies an optional numerical maximum field width
  - a conversion code

The format can be defined using:

<code>%d</code>	Use <code>%d</code> to match a decimal number, which contains an optional + or - sign followed by sequence of characters that include, 0,1,2,3,4,5,6,7,8,9,and <code>_</code> .
<code>%(f, e, or g)</code>	Use <code>%(f, e, or g)</code> to match a floating point number, which contains a string of digits that include, 0,1,2,3,4,5,6,7,8,9. Optionally, you can use a decimal point character ( <code>.</code> ), an exponent part including e or E, and an optional sign before the string of digits.
<code>%r</code>	Use <code>%r</code> to match a 'real' number, using the scale factors defined in LRM2.6.2.
<code>%c</code>	Use <code>%c</code> to match a single character.
<code>%s</code>	Use <code>%s</code> to matches a string, which is a sequence of nonwhite space characters.
<code>%[ ]</code>	Use <code>%[ ]</code> to specify the characters that need to be skipped. The * character should be specified before the characters to be skipped as shown: <pre>\$sscanf(source, "%*[a-z] %s", str1);</pre> The above setting will skip a string containing a,b,c,...,z , and read the next string <code>str1</code> .

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

`%\n` Use of `%\n` displays an error because `\n` is not used in case of strings.

The following is an example of using the `$sscanf` function:

```
source = "a5bs -15.0 2 a anb\n";
    re = $sscanf(source, "%s %e %d %c %s", str1, r, i, il, str2);
    $display("re = %d\n str1 = %s\n r = %e\n i = %d\n il = %c\n str2 = %s\n",
re, str1, r, i, il, str2);
```

In the above example:

```
re = 5
str1 = a5bs
r = -1.500000e+01
i = 2
il = a
str2 = anb
```

### **atoi**

Use the `atoi` function to convert a string to an integer.

```
atoi_function ::=
    int_as_str.atoi()
```

For example,

```
inputstr1 = "456";
str1 = inputstr1.atoi(); // Returns 456
inputstr2 = "99.9";
str2 = inputstr2.atoi(); // Returns 99
inputstr3 = "cj0";
str3 = inputstr3.atoi(); // Causes an error to be reported
```

### **atoreal**

Use the `atoreal` function to convert a string to a real.

```
atoreal_function ::=
    real_as_str.atoreal()
```

For example:

```
inputstr1 = "3.142";
r1 = inputstr1.atoreal(); // Returns 3.142
inputstr2 = "66e6";
r2 = inputstr2.atoreal(); // Returns 6.6e7
inputstr3 = "Gm";
r3 = inputstr3.atoreal(); // Causes an error to be reported
```

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

#### getc

Use the `getc` function to obtain the ASCII code of the first character of a string.

```
getc_function ::=  
    character.getc()
```

Note that the data type of *character* is string. If *character* is an empty string or is undefined, an error is reported. If *character* is a multiple character string, a warning is issued.

For example:

```
inputstr1 = " ";  
code1 = inputstr1.getc(); // Returns 32  
inputstr2 = "6";  
code2 = inputstr2.getc(); // Returns 54  
inputstr3 = "67";  
code3 = inputstr3.getc(); // Returns 54 and causes  
    //a warning about using a multiple  
    //character string as an argument  
inputstr4 = "G";  
code4 = inputstr4.getc(); // Returns 71  
inputstr5 = "";  
code5 = inputstr5.getc(); // Causes an error to be reported
```

#### len

Use the `len` function to determine the number of characters in a string.

```
len_function ::=  
    str.len()
```

For example,

```
inputstr1 = "a short string";  
len1 = inputstr1.len(); // returns 14
```

#### shdl\_strchr

Use the `shdl_strchr` function to find where the first instance of a character occurs in a string.

```
shdl_strchr_function ::=  
    shdl_strchr (input_string, character)
```

The data type of *character* is string. `shdl_strchr` returns the first position in *input\_string* where *character* is found. The function returns -1 if *character* is not found in *input\_string*. An error is reported if either *input\_string* or *character* is undefined. If *character* is an empty string, an error is also reported. If *character* is a multiple-character string, a warning is issued.

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

To use this function, you must use a ``include` statement to include the `shdl_strings.vams` file in the module that uses the function, just before the `analog` statement.

For example

```
`include "shdl_strings.vams"
...
pos1 = shdl_strchr("ABCDEFGHI", "E"); // Returns 4
pos2 = shdl_strchr("abcdefghi", "C"); // Returns -1
```

### **shdl\_strcspn**

Use the `shdl_strcspn` function to count sequences of characters in *input\_string* that are not in a particular set of characters.

```
shdl_strcspn_function ::=
    shdl_strcspn(input_string, span_set)
```

The function returns the number of continuous characters from the start of *input\_string* that are not in *span\_set*. If either *input\_string* or *span\_set* is an undefined string, an error is reported. An error is also reported if *span\_set* is an empty string.

To use this function, you must use a ``include` statement to include the `shdl_strings.vams` file in the module that uses the function, just before the `analog` statement.

For example:

```
`include "shdl_strings.vams"
...
num1 = shdl_strcspn("cjc=1234.0", "0123456789"); // returns 4
num2 = shdl_strcspn("format=nutmeg", "="); // returns 6
```

### **shdl\_strrchr**

Use the `shdl_strrchr` function to find where the last instance of a character occurs in a string.

```
shdl_strrchr_function ::=
    shdl_strrchr(input_string, character)
```

The data type of *character* is string. `shdl_strrchr` returns the last position in *input\_string* where *character* is found. The function returns `-1` if *character* is not found in *input\_string*. An error is reported if either *input\_string* or *character* is undefined. If *character* is an empty string, an error is also reported. If *character* is a multiple character string, a warning is issued.

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

To use this function, you must use a ``include` statement to include the `shdl_strings.vams` file in the module that uses the function, just before the `analog` statement.

For example:

```
`include "shdl_strings.vams"
...
num1 = shdl_strrchr("first x, last x", "x"); // Returns 14
num2 = shdl_strrchr("abcdefghi", "l"); // Returns -1
```

### **shdl\_strspn**

Use the `shdl_strspn` function to count sequences of a set of characters in a particular string.

```
shdl_strspn_function ::=
    shdl_strspn(input_string, span_set)
```

`shdl_strspn` returns the number of continuous characters from the start of `input_string` that are in `span_set`. If either `input_string` or `span_set` is an undefined string, an error is reported. An error is also reported if `span_set` is an empty string.

To use this function, you must use a ``include` statement to include the `shdl_strings.vams` file in the module that uses the function, just before the `analog` statement.

For example:

```
`include "shdl_strings.vams"
...
num1 = shdl_strspn("1234.0", "0123456789"); // Returns 4
num2 = shdl_strspn("/*comment", "/*"); // Returns 2
```

### **shdl\_strstr**

Use the `shdl_strstr` function to find where the first instance of `substring` occurs in `input_string`.

```
shdl_strstr_function ::=
    shdl_strstr (input_string, substring)
```

The function returns in `input_string` the first position where `substring` is found. `shdl_strstr` returns `-1` if `substring` is not found in `input_string`.

To use this function, you must use a ``include` statement to include the `shdl_strings.vams` file in the module that uses the function, just before the `analog` statement.

## Cadence Verilog-A Language Reference

### Operators for Analog Blocks

---

For example:

```
\include "shdl_strings.vams"  
...  
pos1 = shdl_strstr("a little string in a big string", "little");//Returns 2  
pos2 = shdl_strstr("filename = myfile", "herfile"); // Returns -1
```

### **substr**

Use the `substr` function to extract a portion of a string.

```
substr_function ::=  
    str.substr(start_pos, end_pos)
```

This function returns the substring of *str* starting at position *start\_pos* of *str* up to and including *end\_pos*. For example:

```
string1 = "Vds =";  
substr1 = string1.substr(0,2); // returns "Vds"  
string2 = "File=myfile"  
substr2 = string2.substr(5,string2.len()-1);//returns "myfile"
```

---

## Built-In Mathematical Functions

---

This chapter describes the mathematical functions provided by the Cadence® Verilog®-A language. These functions include

- [Standard Mathematical Functions](#) on page 112
- [Trigonometric and Hyperbolic Functions](#) on page 113
- [Controlling How Math Domain Errors Are Handled](#) on page 113

Because the simulator uses differentiation to evaluate expressions, Cadence recommends that you use only mathematical expressions that are continuously differentiable. To prevent run-time domain errors, make sure that each argument is within a function's domain.

## Standard Mathematical Functions

These are the standard mathematical functions supported by Verilog-A. The operands must be integers or real numbers.

Function	Verilog Function Style	Description	Domain	Returned Value
<code>abs(x)</code>		Absolute	All $x$	Integer, if $x$ is integer; otherwise, real
<code>ceil(x)</code>	<code>\$ceil(x)</code>	Smallest integer larger than or equal to $x$	All $x$	Real
<code>exp(x)</code>	<code>\$exp(x)</code>	Exponential. See also " <a href="#">Limited Exponential Function</a> " on page 161.		Real
<code>floor(x)</code>	<code>\$floor(x)</code>	Largest integer less than or equal to $x$	All $x$	Real
<code>ln(x)</code>	<code>\$ln(x)</code>	Natural logarithm	$x > 0$	Real
<code>log(x)</code>	<code>\$log10(x)</code>	Decimal logarithm	$x > 0$	Real
<code>max(x, y)</code>		Maximum	All $x$ , all $y$	Integer, if $x$ and $y$ are integers; otherwise, real
<code>min(x, y)</code>		Minimum	All $x$ , all $y$	Integer, if $x$ and $y$ are integers; otherwise, real
<code>pow(x, y)</code>	<code>\$pow(x, y)</code>	Power of ( $x^y$ )	All $y$ , if $x > 0$ $y > 0$ , if $x = 0$ $y$ integer, if $x < 0$	Real
<code>sqrt(x)</code>	<code>\$sqrt(x)</code>	Square root	$x \geq 0$	Real



## Trigonometric and Hyperbolic Functions

These are the trigonometric and hyperbolic functions supported by Verilog-A. The operands must be integers or real numbers. The simulator converts operands to real numbers if necessary.

The trigonometric and hyperbolic functions require operands specified in radians.

Function	Verilog Function Style	Description	Domain
<code>sin(x)</code>	<code>\$sin(x)</code>	Sine	All $x$
<code>cos(x)</code>	<code>\$cos(x)</code>	Cosine	All $x$
<code>tan(x)</code>	<code>\$tan(x)</code>	Tangent	$x \neq n\left(\frac{\pi}{2}\right)$ , $n$ is odd
<code>asin(x)</code>	<code>\$asin(x)</code>	Arc-sine	$-1 \leq x \leq 1$
<code>acos(x)</code>	<code>\$acos(x)</code>	Arc-cosine	$-1 \leq x \leq 1$
<code>atan(x)</code>	<code>\$atan(x)</code>	Arc-tangent	All $x$
<code>atan2(x, y)</code>	<code>\$atan2(x, y)</code>	Arc-tangent of $x/y$	All $x$ , all $y$
<code>hypot(x, y)</code>	<code>\$hypot(x, y)</code>	$\text{Sqrt}(x^2 + y^2)$	All $x$ , all $y$
<code>sinh(x)</code>	<code>\$sinh(x)</code>	Hyperbolic sine	All $x$
<code>cosh(x)</code>	<code>\$cosh(x)</code>	Hyperbolic cosine	All $x$
<code>tanh(x)</code>	<code>\$tanh(x)</code>	Hyperbolic tangent	All $x$
<code>asinh(x)</code>	<code>\$asinh(x)</code>	Arc-hyperbolic sine	All $x$
<code>acosh(x)</code>	<code>\$acosh(x)</code>	Arc-hyperbolic cosine	$x \geq 1$
<code>atanh(x)</code>	<code>\$atanh(x)</code>	Arc-hyperbolic tangent	$-1 \leq x \leq 1$

## Controlling How Math Domain Errors Are Handled

To control how math domain errors are handled in Verilog-A modules, you can use the options `ahdldomainerror` parameter in a Spectre control file. This parameter controls how domain (out-of-range) errors in Verilog-A math functions such as `log` or `atan` are handled and determines what kind of message is issued when a domain error is found.

## Cadence Verilog-A Language Reference

### Built-In Mathematical Functions

---

The `ahdldomainerror` parameter format is

*Name* **options** **ahdldomainerror**=*value*

where the syntax items are defined as follows.

*Name*                      The unique name you give to the `options` statement. The Spectre simulator uses this name to identify this statement in error or annotation messages

*value*

## Cadence Verilog-A Language Reference

### Built-In Mathematical Functions

---

none	<p>If a domain error occurs, the simulation continues with the argument of the math function set to the nearest reasonable number to the invalid argument. The simulator does not issue any message.</p> <p>For example, if the <code>`sqrt</code> function encounters a negative value, the simulator resets the argument to 0.0.</p>
warning	<p>If a domain error occurs within a converged and accepted time step, the simulator issues a warning message from the last iteration of the time step that had a domain error. The simulation continues with the argument of the math function set to the nearest reasonable number to the invalid argument. This is the default.</p> <p>For example, if the <code>`sqrt</code> function encounters a negative value, the simulator resets the argument to 0.0.</p>
error	<p>If a domain error occurs within a converged and accepted time step, the simulator issues a message from the last iteration of the time step that had a domain error and the simulation terminates.</p> <p>For example:</p> <pre>Fatal error found by spectre during IC analysis, during transient analysis `mytran'. "acosh.va" 20: r1: negative argument passed to `sqrt()'. (value passed was -1.000000)</pre> <p>This message indicates a problem with the <code>`sqrt</code> function.</p>
warniter	<p>For each iteration that has a domain error, the simulator issues a warning message. The simulation continues with the argument of the math function set to the nearest reasonable number to the invalid argument.</p> <p>For example, if the <code>`sqrt</code> function encounters a negative value, the simulator resets the argument to 0.0.</p>
erroriter	<p>For any iteration that has a domain error, the simulator issues a message such as the following and the simulation terminates.</p> <pre>Fatal error found by spectre during IC analysis, during transient analysis `mytran'. "acosh.va" 20: r1: negative argument passed to `sqrt()'. (value passed was -1.000000)</pre> <p>This message indicates a problem with the <code>`sqrt</code> function.</p>

For example, you might have the following in a Spectre control file so that the simulation terminates after a converged and accepted time step if a domain error occurs.

```
myoption options ahdlldomainerror=error
```

**Cadence Verilog-A Language Reference**  
Built-In Mathematical Functions

---

---

## Detecting and Using Analog Events

---

During a simulation, the simulator generates analog events that you can use to control the behavior of your modules. The simulator generates some of these events automatically at various stages of the simulation. The simulator generates other events in accordance with criteria that you specify. Your modules can detect either kind of event and use the occurrences to determine whether specified statements run.

This chapter discusses the following kinds of events

- [Initial\\_step Event](#) on page 119
- [Final\\_step Event](#) on page 119
- [Cross Event](#) on page 120
- [Above Event](#) on page 122
- [Timer Event](#) on page 124

## Detecting and Using Events

Use the @ operator to run a statement under the control of particular events.

```
event_control_statement ::=
    @ ( event_expr ) statement ;
event_expr ::=
    simple_event [ or event_expr ]
simple_event ::=
    initial_step_event
    | final_step_event
    | cross_event
    | timer_event
```

*statement* is the statement controlled by *event\_expr*. The *statement*:

- Cannot include expressions that use analog operators.
- Cannot be a contribution statement.

*simple\_event* is an event that you want to detect. The behavior depends on the context:

- In the analog context, when, and only when, *simple\_event* occurs, the simulator runs *statement*. Otherwise, *statement* is skipped. The kinds of simple events are described in the following sections.
- In the digital context, processing of the block is prevented until the event expression evaluates to true.

If you want to detect more than one kind of event, you can use the event `or` operator. Any one of the events joined with the event `or` operator causes the simulator to run *statement*. The following fragment, for example, sets `V(out)` to zero or one at the beginning of the analysis and at any time `V(sample)` crosses the value 2.5.

```
analog begin
    @(initial_step or cross(V(sample)-2.5, +1)) begin
        vout = (V(in) > 2.5) ;
    end
    V(out) <+ vout ;
end
```

---

### For information on

### See

---

initial_step_event	<a href="#">“Initial_step Event”</a> on page 119
final_step_event	<a href="#">“Final_step Event”</a> on page 119
cross_event	<a href="#">“Cross Event”</a> on page 120
above_event	<a href="#">“Above Event”</a> on page 122

---

**For information on**

**See**

timer\_event

"Timer Event" on page 124

---

## Initial\_step Event

The simulator generates an `initial_step` event during the solution of the first point in specified analyses, or, if no analyses are specified, during the solution of the first point of every analysis. Use the `initial_step` event to perform an action that should occur only at the beginning of an analysis.

```
initial_step_event ::=
    initial_step [ ( analysis_list ) ]
analysis_list ::=
    analysis_name { , analysis_name }
analysis_name ::=
    "analysis_identifier"
```

If the string in *analysis\_identifier* matches the analysis being run, the simulator generates an `initial_step` event during the solution of the first point of that analysis. If you do not specify `analysis_list`, the simulator generates an `initial_step` event during the solution of the first point, or initial DC analysis, of every analysis.

## Final\_step Event

The simulator generates a `final_step` event during the solution of the last point in specified analyses, or, if no analyses are specified, during the solution of the last point of every analysis. Use the `final_step` event to perform an action that should occur only at the end of an analysis.

```
final_step_event ::=
    final_step [ ( analysis_list ) ]
analysis_list ::=
    analysis_name { , analysis_name }
analysis_name ::=
    "analysis_identifier"
```

If the string in *analysis\_identifier* matches the analysis being run, the simulator generates a `final_step` event during the solution of the last point of that analysis. If you do not specify `analysis_list`, the simulator generates a `final_step` event during the solution of the last point of every analysis.

You might use the `final_step` event to print out the results at the end of an analysis. For example, module `bit_error_rate` measures the bit-error of a signal and prints out the

## Cadence Verilog-A Language Reference

### Detecting and Using Analog Events

---

results at the end of the analysis. (This example also uses the timer event, which is discussed in [“Timer Event”](#) on page 124.)

```
module bit_error_rate (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 ;                               // Initialize the variables
    end
    @(timer(0, period)) begin
        if ((V(in) > thresh) != (V(ref) > thresh))
            errors = errors + 1;                   // Check for errors each period
        bits = bits + 1 ;
    end
    @(final_step)
        $strobe("Bit error rate = %f%%", 100.0 * errors/bits );
end
endmodule
```

## Cross Event

According to criteria you set, the simulator can generate a cross event when an expression crosses zero in a specified direction. Use the `cross` function to specify which crossings generate a cross event.

```
cross_function ::=
    cross (expr1 [ , direction [ , time_tol [ , expr_tol [ , enable ] ] ] )
direction ::=
    +1 | 0 | -1
time_tol ::=
    expr2
expr_tol ::=
    expr3
```

*expr1* is the real expression whose zero crossing you want to detect.

*direction* is an integer expression set to indicate which zero crossings the simulator should detect.

---

<b>If you want to</b>	<b>Then</b>
Detect all zero crossings	Do not specify <i>direction</i> , or set <i>direction</i> equal to 0
Detect only zero crossings where the value is increasing	Set <i>direction</i> equal to +1



## Cadence Verilog-A Language Reference

### Detecting and Using Analog Events

---

#### If you want to

#### Then

Detect only zero crossings where the value is decreasing

Set `direction` equal to -1

---

`time_tol` is a constant expression with a positive value, which is the largest time interval that you consider negligible. The default value is 1.0s, which is large enough that the tolerance is almost always satisfied.

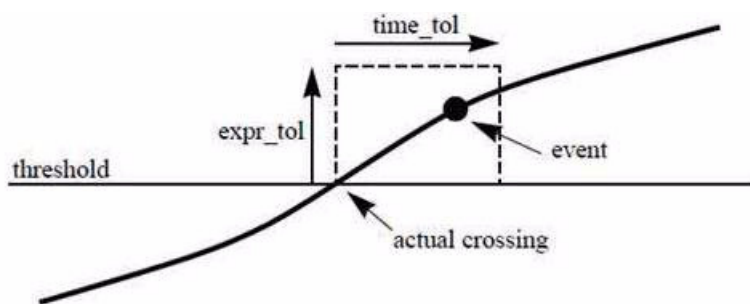
`expr_tol` is a constant expression with a positive value, which is the largest difference that you consider negligible. If you specify `expr_tol`, both it and `time_tol` must be satisfied. If you do not specify `expr_tol`, the simulator uses the default `expr_tol` value of

`1e-9 + reltol*max_value_of_the_signal`

In addition to generating a cross event, the `cross` function also controls the time steps to accurately resolve each detected crossing.

The event shall occur after the threshold crossing, and while the signal remains in the box defined by actual crossing of `time_tol` and `expr_tol`.

**Figure 8-1 Timing of event relative to threshold crossing**



If the time difference between the actual crossing and the event is larger than `time_tol`, the simulator will reduce the time step.

If the difference between threshold and the value of the event is larger than `expr_tol`, the simulator will also reduce the time step.

Set `time_tol` to 1s means only `expr_tol` will limit the time step.

`enable` is an expression, which, if specified as zero, causes the simulator not to generate any cross events or control the time step.

The `cross` function is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

The following example illustrates how you might use the `cross` function and event. The `cross` function generates a cross event each time the sample voltage increases through the value 2.5. `expr_tol` is specified as the `abstol` associated with the potential nature of the net `sample`.

```
module samphold (in, out, sample) ;
output out ;
input in, sample ;
electrical in, out, sample ;
real hold ;

analog begin
    @(cross(V(sample)-2.5, +1, 0.01n, sample.potential.abstol))
        hold = V(in) ;
    V(out) <+ transition(hold, 0, 10n) ;
end
endmodule
```

## Above Event

According to criteria you set, the simulator can generate an above event when an expression becomes greater than or equal to zero. Use the `above` function to specify when the simulator generates an above event. An above event can be generated and detected during initialization. By contrast, a cross event can be generated and detected only after at least one transient time step is complete.

The `above` function is a Cadence language extension.

```
above_function ::=
    above (expr1 [ , time_tol [ , expr_tol [ , enable ] ] ] )
time_tol ::=
    expr2
expr_tol ::=
    expr3
```

*expr1* is a real expression whose value is to be compared with zero.

*time\_tol* is a constant real expression with a positive value, which is the largest time interval that you consider negligible.

*expr\_tol* is a constant real expression with a positive value, which is the largest difference that you consider negligible. If you specify *expr\_tol*, both it and *time\_tol* must be satisfied. If you do not specify *expr\_tol*, the simulator uses the value of its own `reltol` parameter.

## Cadence Verilog-A Language Reference

### Detecting and Using Analog Events

---

During a transient analysis, after  $t = 0$ , the `above` function behaves the same as a `cross` function with the following specification.

```
cross(expr1 , 1 , time_tol, expr_tol )
```

During a transient analysis, the `above` function controls the time steps to accurately resolve the time when `expr1` rises to zero or above.

`enable` is an expression, which, if specified as zero, causes the simulator not to generate any `above` event or control the time step.

The `above` function is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

The following example, taken from the sample library, illustrates how to use the `above` function.

```
module and_gate(vin1, vin2, vout);
input vin1, vin2;
output vout;
electrical vin1, vin2, vout;
parameter real vlogic_high = 5;
parameter real vlogic_low = 0;
parameter real vtrans = 1.4;
parameter real tdel = 2u from [0:inf);
parameter real trise = 1u from (0:inf);
parameter real tfall = 1u from (0:inf);

    real vout_val;
    integer logic1, logic2;
analog begin
    @ ( initial_step ) begin
        if (vlogic_high < vlogic_low) begin
            $display("Range specification error.  vlogic_high = (%E) less than vlogic_low = (%E).\n", vlogic_high, vlogic_low );
            $finish;
        end
        if (vtrans > vlogic_high || vtrans < vlogic_low) begin
            $display("Inconsistent $threshold specification w/logic family.\n");
        end
    end
    @ (above(V(vin1) - vtrans)) logic1 = 1;
    @ (above(vtrans - V(vin1))) logic1 = 0;

    @ (above(V(vin2) - vtrans)) logic2 = 1;
    @ (above(vtrans - V(vin2))) logic2 = 0;

    //
    // define the logic function.
    //
    vout_val = (logic1 && logic2) ? vlogic_high : vlogic_low;
    V(vout) <+ transition( vout_val, tdel, trise, tfall);
end
endmodule
```

## Timer Event

According to criteria you set, the simulator can generate a timer event at specified times during a simulation. Use the `timer` function to specify when the simulator generates a timer event.

Do not use the `timer` function inside conditional statements.

```
timer_function ::=  
    timer ( start_time [ , period [ , timetol [ , enable ] ] ] )
```

`start_time` is a dynamic expression specifying an initial time. The simulator places a first time step at, or just beyond, the `start_time` that you specify and generates a timer event.

`period` is a dynamic expression specifying a time interval. The simulator places time steps and generates events at each multiple of `period` after `start_time`.

`timetol` is a constant expression specifying how close a placed time point must be to the actual time point.

`enable` is an expression, which, if specified as zero, causes the simulator not to generate any timer event. However, when `enable` is set to a non-zero value, the simulator starts generating the timer event at the specified times.

The module `squarewave`, below, illustrates how you might use the `timer` function to generate timer events. In `squarewave`, the output voltage changes from positive to negative or from negative to positive at every time interval of `period/2`.

```
module squarewave (out)  
output out ;  
electrical out ;  
parameter period = 1.0 ;  
integer x ;  
analog begin  
    @(initial_step) x = 1 ;  
    @(timer(0, period/2)) x = -x ;  
    V(out) <+ transition(x, 0.0, period/100.0 ) ;  
end  
endmodule
```

---

## Simulator Functions

---

This chapter describes the Cadence® Verilog®-A language simulator functions. The simulator functions let you access information about a simulation and manage the simulation's current state. You can also use the simulator functions to display and record simulation results.

For information about using simulator functions, see

- [Announcing Discontinuity](#) on page 127
- [Bounding the Time Step](#) on page 129
- [Finding When a Signal Is Zero](#) on page 130
- [Querying the Simulation Environment](#) on page 131
- [Relating a Specific Frequency to a Source Name for RF](#) on page 134
- [Detecting Parameter Overrides](#) on page 135
- [Detecting Port Binding](#) on page 136
- [Obtaining and Setting Signal Values](#) on page 137
- [Determining the Current Analysis Type](#) on page 141
- [Implementing Small-Signal AC Sources](#) on page 143
- [Implementing Small-Signal Noise Sources](#) on page 144
- [Generating Random Numbers](#) on page 146
- [Generating Random Numbers in Specified Distributions](#) on page 149
- [Interpolating with Table Models](#) on page 156

For information on analog operators and filters, see

- [Limited Exponential Function](#) on page 161
- [Time Derivative Operator](#) on page 162

## Cadence Verilog-A Language Reference

### Simulator Functions

---

- [Time Integral Operator](#) on page 162
- [Circular Integrator Operator](#) on page 164
- [Delay Operator](#) on page 167
- [Transition Filter](#) on page 168
- [Slew Filter](#) on page 171
- [Implementing Laplace Transform S-Domain Filters](#) on page 173
- [Implementing Z-Transform Filters](#) on page 178

For descriptions of functions used to control input and output, see:

- [Displaying Results](#) on page 183
- [Working with Files](#) on page 191

For descriptions of functions used to control the simulator, see

- [Simulator Control Functions](#) on page 198

For a description of the `$pwr` function, which is used to specify power consumption in a module, see

- [Specifying Power Consumption](#) on page 188
- [Indicating Non-linearities to the Simulator](#) on page 189

For information on using user-defined functions in the Verilog-A language, see

- [Declaring an Analog User-Defined Function](#) on page 203
- [Calling a User-Defined Analog Function](#) on page 207
- [Calling functions implemented in C](#) on page 208

## Announcing Discontinuity

Use the `$discontinuity` function to tell the simulator about a discontinuity in signal behavior.

```
discontinuity_function ::=  
    $discontinuity[ (constant_expression) ]
```

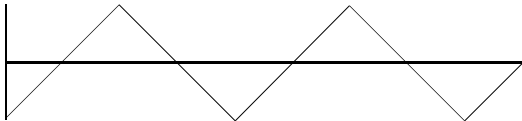
*constant\_expression*, which must be zero or a positive integer, is the degree of the discontinuity. For example, `$discontinuity`, which is equivalent to `$discontinuity(0)`, indicates a discontinuity in the equation, and `$discontinuity(1)` indicates a discontinuity in the slope of the equation.

You do not need to announce discontinuities created by switch branches or built-in functions such as `transition` and `slew`.

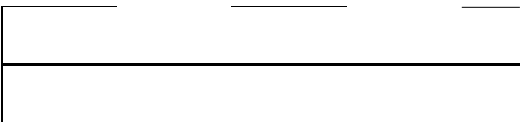
Be aware that using the `$discontinuity` function does not guarantee that the simulator will be able to handle a discontinuity successfully. If possible, you should avoid discontinuities in the circuits you model.

The following example shows how you might use the `$discontinuity` function while describing the behavior of a source that generates a triangular wave. As the [Triangular Wave](#) figure on page 127 shows, the triangular wave is continuous, but as the [Triangular Wave First Derivative](#) figure on page 127 shows, the first derivative of the wave is discontinuous.

### Triangular Wave



### Triangular Wave First Derivative



The module `trisource` describes this triangular wave source.

```
module trisource (vout) ;  
    output vout ;  
    voltage vout ;  
    parameter real wavelength = 10.0, amplitude = 1.0 ;  
    integer slope ;  
    real wstart ;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
analog begin
  @(timer(0, wavelength)) begin
    slope = +1 ;
    wstart = $abstime ;
    $discontinuity (1);          // Change from neg to pos slope
  end
  @(timer(wavelength/2, wavelength)) begin
    slope = -1 ;
    wstart = $abstime ;
    $discontinuity (1);          // Change from pos to neg slope
  end
  V(vout) <+ amplitude * slope * (4 * ($abstime - wstart) / wavelength-1) ;
end
endmodule
```

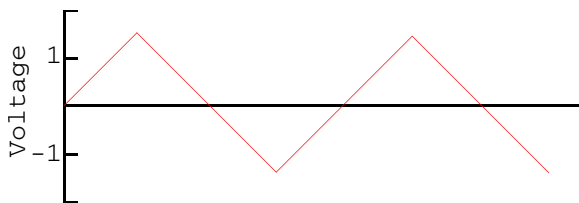
The two `$discontinuity` functions in `trisource` tell the simulator about the discontinuities in the derivative. In response, the simulator uses analysis techniques that take the discontinuities into account.

The module `relay`, as another example, uses the `$discontinuity` function while modeling 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) - 1, 0, 0.01n, pin.potential.abstol)) $discontinuity(0);
  if (V(pin, nin) >= 1)
    I(c1, c2) <+ V(c1, c2) / r ;
  else
    I(c1, c2) <+ 0 ;
end
endmodule
```

The `$discontinuity` function in `relay` tells the simulator that there is a discontinuity in the current when the voltage crosses the value 1. For example, passing a triangular wave like that shown in the [Relay Voltage](#) figure on page 128 through module `relay` produces the discontinuous current shown in the [Relay Current](#) figure on page 129.

### Relay Voltage





## Relay Current



## Bounding the Time Step

Use the `$bound_step` function to specify the maximum time allowed between adjacent time points during simulation.

```
bound_step_function ::=
    $bound_step ( max_step )
max_step ::=
    constant_expression
```

By specifying appropriate time steps, you can force the simulator to track signals as closely as your model requires. For example, module `sinwave` forces the simulator to simulate at least 50 time points during each cycle.

```
module sinwave (outsig) ;
output outsig ;
voltage outsig ;
parameter real freq = 1.0, ampl = 1.0 ;
analog begin
    V(outsig) <+ ampl * sin(2.0 * `M_PI * freq * $abstime) ;
    $bound_step(0.02 / freq) ; // Max time step = 1/50 period
end
endmodule
```

## Finding When a Signal Is Zero

Use the `last_crossing` function to find out what the simulation time was when a signal expression last crossed zero.

```
last_crossing_function ::=  
    last_crossing ( signal_expression , direction )
```

Set *direction* to indicate which crossings the simulator should detect.

---

<b>If you want to</b>	<b>Then</b>
Detect all crossings	Set <i>direction</i> equal to 0
Detect only crossings where the value is increasing	Set <i>direction</i> equal to +1
Detect only crossings where the value is decreasing	Set <i>direction</i> equal to -1

---

Before the first detectable crossing, the `last_crossing` function returns a negative value.

The `last_crossing` function is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

The `last_crossing` function does not control the time step to get accurate results and uses interpolation to estimate the time of the last crossing. To improve the accuracy, you might want to use the `last_crossing` function together with the `cross` function.

For example, module `period` calculates the period of the input signal, using the `cross` function to resolve the times accurately.

```
module period (in) ;  
input in ;  
voltage in ;  
integer crosscount ;  
real latest, earlier ;  
analog begin  
    @(initial_step) begin  
        crosscount = 0 ;  
        earlier = 0 ;  
    end  
  
    @(cross(V(in), +1)) begin  
        crosscount = crosscount + 1 ;  
        earlier = latest ;  
    end  
    latest = last_crossing(V(in), +1) ;  
    @(final_step) begin
```

```
    if (crosscount < 2)
        $strobe("Could not measure the period.") ;
    else
        $strobe("Period = %g, Crosscount = %d", latest-earlier, crosscount) ;
    end
end
endmodule
```

## Querying the Simulation Environment

Use simulation environment functions to obtain information about the current simulation environment. See the following topics for details:

- [Obtaining the Current Simulation Time](#) on page 131
- [Obtaining the Current Ambient Temperature](#) on page 132
- [Obtaining the Thermal Voltage](#) on page 132
- [Querying the scale, gmin, and iteration Simulation Parameters](#) on page 132

### Obtaining the Current Simulation Time

Use `$abstime` to obtain the current simulation time. See the following topics for more information:

- [\\$abstime Function](#) on page 131
- [Using \\$abstime for RF Analysis](#) on page 131

### **\$abstime Function**

Use the `$abstime` function to obtain the current simulation time in seconds.

```
abstime_function ::=
    $abstime
```

### Using \$abstime for RF Analysis

For Spectre RF simulation, you can use a sine or cosine function with `$abstime` to create a periodic source that works in both the time and the frequency domains. You must define the periodic source using a `sin` or `cos` function in order to use it for RF analysis. The argument to these functions must be linear in time and of the following form:

```
cos( expressionA * $abstime + expressionB )
sin( expressionA * $abstime + expressionB )
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

For example, the following module outputs a sinusoidal source whose name is `rf` and whose value is 1G:

```
module example (out);
output out;
electrical out;
parameter string fundname = "rf";
parameter real freq = 1G;
analog begin
    $cds_set_rf_source_info ( fundname, freq);
    V(out) <+ sin ( `M_TWO_PI * freq * $abstime + `M_PI_4 );
end
endmodule
```

See also [“Relating a Specific Frequency to a Source Name for RF”](#) on page 134.

## Obtaining the Current Ambient Temperature

Use the `$temperature` function to obtain the ambient temperature of a circuit in degrees Kelvin.

```
temperature_function ::=
    $temperature
```

## Obtaining the Thermal Voltage

Use the `$vt` function to obtain the thermal voltage,  $(kT/q)$ , of a circuit.

```
vt_function ::=
    $vt [(temp)]
```

`temp` is the temperature, in degrees Kelvin, at which the thermal voltage is to be calculated. If you do not specify `temp`, the thermal voltage is calculated at the temperature returned by the `$temperature` function.

## Querying the scale, gmin, and iteration Simulation Parameters

Use the `$simparam` function to query the value of the `scale`, `gmin`, or `iteration` simulation parameters. The returned value is always a real value.

```
simparam_function ::=
    $simparam ("param" [, expression])
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

*param* is one of the following simulation parameters.

---

Simulation Parameter	Meaning
<code>scale</code>	Scale factor for device instance geometry parameters.
<code>gmin</code>	Minimum conductance placed in parallel with nonlinear branches.
<code>iteration</code>	Iteration number of the analog solver.

---

*expression* is an expression whose value is returned if *param* is not recognized.

For example, to return the value of the simulation parameter named `gmin`, you might code

```
$strobe("gmin = %e", $simparam("gmin")) ;
```

To specify that a value of 2.0 is to be returned when the specified simulation parameter is not found, you might code

```
$strobe("gmin = %e", $simparam("gmin", 2.0)) ;
```

## Probing of values within a sibling instance during simulation

### Dynamic simulation probe function

Verilog-AMS HDL supports Dynamic Simulation Probe function `$simprobe()` that allows probing of values within a sibling instance during simulation.

`$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 looks 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 then queries 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, an error is displayed. If the optional expression is supplied, its value is displayed.

The intended use of this function is to allow dynamic monitoring of instance quantities.

## 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` probes the `quant` in instance `inst`. If its value becomes larger than `threshold`, an error is raised and the simulation is stopped.

```
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` keeps 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, the instance `amonitor` generates the error message and stops the simulation

## Relating a Specific Frequency to a Source Name for RF

For Spectre RF simulation, use the `$cds_set_rf_source_info` function to relate a specific frequency to a source name as follows:

```
$cds_set_rf_source_info( t_sourceName, n_frequencyValue )
```

The arguments of this function are as follows:

<i>t_sourceName</i>	Name of source Valid Values: Any string that corresponds to a valid RF source name
<i>n_frequencyValue</i>	Single frequency value Valid Values: A double-precision floating point number

For example:

```
module example (out);
output out;
electrical out;
parameter string fundname = "rf";
parameter real freq = 1G;
analog begin
    $cds_set_rf_source_info ( fundname, freq );
    ...
end
endmodule
```

## Detecting Parameter Overrides

Use the `$param_given` function to determine whether a parameter value was obtained from the default value in its declaration statement or if that value was overridden.

```
$param_given_function ::=  
    $param_given(module_parameter_identifier)
```

*module\_parameter\_identifier* is the parameter for which it is determined whether the value was overridden. The return value of the function is 1 if the specified parameter was overridden by a module instance parameter value assignment. The return value is 0 otherwise.

You can use the `$param_given` function in a `genvar` expression.

For example, the following fragment allows the code to behave differently when `tdevice` has the default value set by the declaration statement and when the value is actually set by an override.

```
if ($param_given(tdevice))
    temp = tdevice + `P_CELSIUS0;
else
    temp = $temperature;
```

## Detecting Port Binding

Use the `$port_connected` function to determine whether a connection was specified for a port.

```
$port_connected_function ::=  
    $port_connected(port_identifier)
```

Module ports need 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 is one (1) if the port was connected to a net (by order, or by name) when the module was instantiated, and zero (0) otherwise.

In the following example, `$port_connected()` is used to skip the transition filter for unconnected nodes. In the module `twoclk`, the instances of `myclk` have connections only for their `vout_q` ports, and therefore 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 ultimately not used in the circuit; however, the filter for `vout_q` of `clk2` is implemented, because `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_q2;

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
```

## Obtaining and Setting Signal Values

Use the access functions to obtain or set the signal values.

```
access_function_reference ::=
    bvalue
    | pvalue
bvalue ::=
    access_identifier ( analog_signal_list )
analog_signal_list ::=
    branch_identifier
    | array_branch_identifier [ genvar_expression ]
    | net_or_port_scalar_expression
    | net_or_port_scalar_expression , net_or_port_scalar_expression
net_or_port_scalar_expression ::=
    net_or_port_identifier
    | vector_net_or_port_identifier [ genvar_expression ]
pvalue ::=
    flow_access_identifier (port_identifier,port_identifier)
```

Access functions in Verilog-A take their names from the discipline associated with a node, port, or branch. Specifically, the access function names are defined by the access attributes specified for the discipline's natures.

For example, the `electrical` discipline, as defined in the standard definitions, uses the nature `Voltage` for potential. The nature `Voltage` is defined with the access attribute

## Cadence Verilog-A Language Reference

### Simulator Functions

---

equal to  $v$ . Consequently, the access function for electrical potential is named  $v$ . For more information, see the files installed in `your_install_dir/tools/spectre/etc/ahdl`.

To set a voltage, use the  $v$  access function on the left side of a contribution statement.

```
V(out) <+ I(in) * Rparam ;
```

To obtain a voltage, you might use the  $v$  access function as illustrated in the following fragment.

```
I(c1, c2) <+ V(c1, c2) / r ;
```

You can apply access functions only to scalars or to individual elements of a vector. The scalar element of a vector is selected with an index. For example,  $v(in[1])$  accesses the voltage `in[1]`.

To see how you can use access functions, consult the “Access Function Formats” table. In the table, `b1` refers to a branch, `n1` and `n2` refer to either nodes or ports, and `p1` refers to a port. To make the example concrete, the branches, nodes, and ports used in the table 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). Access functions for other disciplines have different names, but you use them in the same ways. For example, `MMF` is the access function for potential in the `magnetic` discipline.

#### Access Function Formats

---

Format	Effect
$V(b1)$	Accesses the potential across branch <code>b1</code>
$V(n1)$	Accesses the potential of <code>n1</code> relative to ground
$V(n1, n2)$	Accesses the potential difference on the unnamed branch between <code>n1</code> and <code>n2</code>
$I(b1)$	Accesses the current on branch <code>b1</code>
$I(n1)$	Accesses the current flowing from <code>n1</code> to ground
$I(n1, n2)$	Accesses the current flowing on the unnamed branch between <code>n1</code> and <code>n2</code> ; node <code>n1</code> and node <code>n2</code> cannot be the same node
$I(<p1>)$	Accesses the current flow into the module through port <code>p1</code> . This format accesses the port branch associated with port <code>p1</code> .

---

You can use a port access to monitor the flow. In the following example, the simulator issues a warning if the total diode current becomes too large.

```
module diode (a, c) ;  
  electrical a, c ;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
branch (a, c) diode, cap ;
parameter real is=1e-14, tf=0, cjo=0, imax=1, phi=0.7 ;
analog begin
  I(diode) <+ is*(limexp(V(diode)/$vt) -1) ;
  I(cap) <+ ddt(tf*I(diode) - 2 * cjo * sqrt(phi * (phi * V(cap)))) ;
  if (I(<a>) > imax) // Checks current through port
    $strobe( "Warning: diode is melting!" ) ;
end
endmodule
```

## Accessing Attributes

Use the hierarchical referencing operator to access the attributes for a node or branch.

```
attribute_reference ::=
    node_identifier.pot_or_flow.attribute_identifier
pot_or_flow ::=
    potential
    | flow
```

*node\_identifier* is the node or branch whose attribute you want to access.

*attribute\_identifier* is the attribute you want to access.

For example, the following fragment illustrates how to access the abstol values for a node and a branch.

```
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) ; // Access abstol for node
    I(cap) <+ c*ddt(V(cap), n1.potential.abstol) ; // Access abstol for branch
end
```

## Analysis-Dependent Functions

The analysis-dependent functions change their behavior according to the type of analysis being performed. See the following topics for more information:

- [Determining the Current Analysis Type](#) on page 141
- [Implementing Small-Signal AC Sources](#) on page 143
- [Implementing Small-Signal Noise Sources](#) on page 144

### Determining the Current Analysis Type

Use the `analysis` function to determine whether the current analysis type matches a specified type. By using this function, you can design modules that change their behavior during different kinds of analyses.

```
analysis ( analysis_list )  
analysis_list ::=  
    analysis_name { , analysis_name }  
analysis_name ::=  
    "analysis_type"
```

`analysis_type` is one of the following analysis types.

### Analysis Types and Descriptions

---

Analysis Type	Analysis Description
<code>ac</code>	AC analysis
<code>all</code>	All analysis types
<code>check</code>	Check parameter values
<code>dc</code>	OP or DC analysis
<code>dcmatch</code>	DC device matching analysis
<code>envlp</code>	Envelope following analysis
<code>harmonic</code>	Harmonic balance analysis
<code>ic</code>	Initial conditions
<code>info</code>	Circuit information
<code>noise</code>	Noise analysis
<code>pac</code>	Periodic AC (PAC) analysis

## Cadence Verilog-A Language Reference

### Simulator Functions

---

#### Analysis Types and Descriptions, *continued*

---

Analysis Type	Analysis Description
<code>pdisto</code>	Periodic distortion analysis
<code>pnoise</code>	Periodic noise analysis
<code>psp</code>	Periodic S-parameter analysis
<code>pss</code>	Periodic steady-state analysis
<code>pxf</code>	Periodic transfer function analysis
<code>pz</code>	Pole-zero analysis
<code>qpac</code>	Quasi-periodic AC analysis
<code>qpnoise</code>	Quasi-periodic noise analysis
<code>qpsp</code>	Quasi-periodic S-parameter analysis
<code>qpss</code>	Quasi-periodic steady state analysis
<code>qpxf</code>	Quasi-periodic transfer function analysis
<code>sp</code>	S-parameter analysis
<code>static</code>	Any equilibrium point calculation, including a DC analysis as well as those that precede another analysis, such as the DC analysis that precedes an AC or noise analysis, or the initial-condition analysis that precedes a transient analysis
<code>stb</code>	Stability analysis
<code>tdr</code>	Time-domain reflectometer analysis
<code>tran</code>	Transient analysis
<code>xf</code>	Transfer function analysis

---

## Cadence Verilog-A Language Reference

### Simulator Functions

The following table describes the values returned by the `analysis` function for some of the commonly used analyses. A return value of 1 represents `TRUE` and a value of 0 represents `FALSE`.

Argument	Simulator Analysis Type							
	DC	TRAN		AC		NOISE		
		OP	TRAN	OP	AC	OP	AC	
<code>static</code>	1	1	0	1	0	1	0	
<code>ic</code>	0	1	0	0	0	0	0	
<code>dc</code>	1	0	0	0	0	0	0	
<code>tran</code>	0	1	1	0	0	0	0	
<code>ac</code>	0	0	0	1	1	0	0	
<code>noise</code>	0	0	0	0	0	1	1	

You can use the `analysis` function to make module behavior dependent on the current analysis type.

```
if (analysis("dc", "ic"))
    out = ! V(in) > 0.0 ;
else
    @(cross (V(in),0)) out = ! out
V(out) <+ transition (out, 5n, 1n, 1n) ;
```

### Implementing Small-Signal AC Sources

Use the `ac_stim` function to implement a sinusoidal stimulus for small-signal analysis.

```
ac_stim ( [ "analysis_type" [ , mag [ , phase]] ] )
```

`analysis_type`, if you specify it, must be one of the analysis types listed in the [Analysis Types and Descriptions](#) table on page 141. The default for `analysis_type` is `ac`. The `mag` argument is the magnitude, with a default of 1. `phase` is the phase in radians, with a default of 0.

The `ac_stim` function models a source with magnitude `mag` and phase `phase` only during the `analysis_type` analysis. During all other small-signal analyses, and during large-signal analyses, the `ac_stim` function returns 0.

## Implementing Small-Signal Noise Sources

Verilog-A provides three functions to support noise modeling during small-signal analyses:

- `white_noise` function
- `flicker_noise` function
- `noise_table` function

### White\_noise Function

Use the `white_noise` function to generate white noise, noise whose current value is completely uncorrelated with any previous or future values.

**`white_noise( PSD [ , "name" ] )`**

*PSD* is the power spectral density of the source where *PSD* is specified in units of  $A^2/Hz$  or  $V^2/Hz$ .

*name* is a label for the noise source. The simulator uses *name* to identify the contributions of noise sources to the total output noise. The simulator combines into a single source all noise sources with the same name from the same module instance.

The `white_noise` function is active only during small-signal noise analyses and returns 0 otherwise.

For example, you might include the following fragment in a module describing the behavior of a diode.

```
I(diode) <+ white_noise(2 * `P_Q * Id, "shot" ) ;
```

For a resistor, you might use a fragment like the following.

```
V(res) <+ white_noise(4 * `P_K * $temperature * rs, "thermal");
```

### flicker\_noise Function

Use the `flicker_noise` function to generate pink noise that varies in proportion to:

$$1/f^{\text{exp}}$$

The syntax for the `flicker_noise` function is

**`flicker_noise( power, exp [ , "name" ] )`**

*power* is the power of the source at 1 Hz.



## Cadence Verilog-A Language Reference

### Simulator Functions

---

*name* is a label for the noise source. The simulator uses *name* to identify the contributions of noise sources to the total output noise. The simulator combines into a single source all noise sources with the same name from the same module instance.

The `flicker_noise` function is active only during small-signal noise analyses and returns 0 otherwise.

For example, you might include the following fragment in a module describing the behavior of a diode:

```
I(diode) <+ flicker_noise( kf * pow(abs(I(diode)),af),ef) ;
```

### Noise\_table Function

Use the `noise_table` function to generate noise where the spectral density of the noise varies as a piecewise linear function of frequency.

```
noise_table(vector | string [ , "name" ])
```

The first argument to the `noise_table` function is a vector or a string.

*vector* is an array containing pairs of real numbers. The first number in each pair represents the frequency in hertz and the second number is the power at that frequency.

*string* is an input file containing frequency and power values specified in pairs. Each pair is specified in a new line and is separated by space(s) or tab(s). You can comment a line by inserting a comment (#) at the beginning or end of the line. The input file can only be a text file and the values must be real numbers or integers. In addition, each frequency value must be unique.

It is recommended that you specify each noise pair in the order of ascending frequency. However, if required, the simulator internally sorts the pairs in the order of ascending frequency.

The `noise_table` function uses the linear interpolation method to calculate the spectral density for each frequency. The power associated with the lowest frequency specified in the table is taken as the associated power for all frequencies below the smallest frequency. Similarly, the power associated with the highest frequency specified in the table is taken as the associated power for all frequencies above the highest frequency.

The following is an example of the input file:

```
# noise_table_input.tbl
# Example of input file format for noise_table
#
# freq          pwr
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
1.0e0      1.348720e-23
1.0e1      5.482480e-23
1.0e2      7.135790e-23
1.0e3      1.811153e-22
1.0e4      3.726892e-22
1.0e5      7.412361e-22
1.0e6      2.172611e-21
# End
```

*name* is a label for the noise source. The simulator uses *name* to identify the contributions of noise sources to the total output noise. The simulator combines into a single source all noise sources with the same name from the same module instance.

The `noise_table` function is active only during small-signal noise analyses and returns 0 otherwise.

For example, you might include the following fragment in an analog block:

```
V(p,n) <+ noise_table({1,2,100,4,1000,5,1000000,6}, "noitab");
```

In this example, the power at every frequency lower than 1 is assumed to be 2; the power at every frequency above 1000000 is assumed to be 6.

## Generating Random Numbers

Use the `$random` and `$arandom` functions to generate a signed integer, 32-bit, pseudorandom number.

### `$random`

```
$random [ ( seed ) ] ;
```

*seed* is a reg, integer, or time variable used to initialize the function. The seed provides a starting point for the number sequence and allows you to restart at the same point. If, as Cadence recommends, you use *seed*, you must assign a value to the variable before calling the `$random` function.

The `$random` function generates a new number every time step.

Individual `$random` statements with different seeds generate different sequences, and individual `$random` statements with the same seed generate identical sequences.

The following code fragment uses the absolute value function and the modulus operator to generate integers between 0 and 99.

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
// There is a 5% chance of signal loss.
module randloss (pinout) ;
electrical pinout ;
integer randseed, randnum;
analog begin
    @ (initial_step) begin
        randseed = 123 ;    // Initialize the seed just once
    end
    randnum = abs($random(randseed) % 100) ;
    if (randnum < 5)
        V(pinout) <+ 0.0 ;
    else
        V(pinout) <+ 3.0 ;
end // of analog block
endmodule
```

### **\$arandom**

The `$arandom` function is similar to the `$random` function. The two functions differ in the arguments they accept. The `$arandom` is upwardly compatible with `$random`, that is, the `$arandom` function can accept the same arguments as `$random` and has the same behavior. The syntax for `$arandom` function is shown below.

```
$arandom [ ( seed[, string] ) ] ;
```

The `seed` argument can be a reg, integer, or time variable. In addition, it can be a parameter or a constant, this allows the `$arandom` function to be used for parameter initialization. In order to get different random values when the `seed` argument is a parameter, you can override the parameter.

The `string` argument is an optional argument and can be `global` or `instance`. It provides support for Monte Carlo analysis. If it is set to `global` then one value is generated for each Monte Carlo trial. If it is set to `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.

The Monte Carlo analysis refers to the statistics blocks where statistical distributions and correlations of netlist parameters are specified. For each trial of the Monte Carlo analysis, new pseudorandom values are generated for the specified netlist parameters (according to their specified distributions) and the list of child analyses are then executed. The statistics blocks allow you to specify batch-to-batch (process) and per-instance (mismatch) variations for netlist parameters. These statistically varying netlist parameters can be referenced by models or instances in the main netlist. For more detail on Monte Carlo analysis and statistics blocks, refer to *Virtuoso Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide*. In Verilog-A models, you can use the `$arandom` function to achieve "statistics blocks" (process and mismatch) by setting the `string` argument of this function to `global` or `instance`. In addition, you can use the `$rdist_uniform`, `$dist_uniform`,

## Cadence Verilog-A Language Reference

### Simulator Functions

---

`$rdist_normal`, `$dist_noram1`, `$rdist_exponential`, `$dist_exponential`, `$rdist_poisson`, `$dist_poisson`, `$rdist_chi_square`, `$dist_chi_square`, `$rdist_t`, `$dist_t`, `$rdist_erlang` and `$dist_erlang` functions to achieve the "statistics blocks" (process and mismatch ).

The following example shows how to use the `$arandom` function. Because the parameter `val_global` is obtained from `$arandom` with the string `global`, and the parameter `val_instance` is obtained from `$arandom` with the string `instance`, the instances `INST_RES_0`, `INST_RES_1`, and `INST_RES_2` will get the same value of `val_global` and get different values of `val_instance`.

```
module res(va, vb);
  inout va, vb;
  electrical va, vb;
  parameter real r = 1;
  parameter integer val_global = $arandom(1, "global") % 100;
  parameter integer val_instance = $arandom(2, "instance") % 100;

  analog begin

    @(initial_step) begin
      $strobe("\nval_global = %d val_instance = %d", val_global, val_instance);
    end

    I(va, vb) <+ V(va, vb)/(r*val_instance);

  end
endmodule

module res_mul(vp, vn);
  inout vp, vn;
  electrical vp, vn;

  res #(.r(1k)) INST_RES_0 (vp, vn); // INST_RES_0: val_global = -48 val_instance
= -55
  res #(.r(1k)) INST_RES_1 (vp, vn); // INST_RES_1: val_global = -48 val_instance
= 65
  res #(.r(1k)) INST_RES_2 (vp, vn); // INST_RES_2: val_global = -48 val_instance
= 48
endmodule
```

## Generating Random Numbers in Specified Distributions

Verilog-A provides functions that generate random numbers in the following distribution patterns:

- Uniform
- Normal (Gaussian)
- Exponential
- Poisson
- Chi-square
- Student's T
- Erlang

In releases prior to IC5.0, the functions beginning with `$dist` return real numbers rather than integer numbers. If you need to continue getting real numbers in more recent releases, change each `$dist` function to the corresponding `$rdist` function.

### Uniform Distribution

Use the `$rdist_uniform` function to generate random real numbers (or the `$dist_uniform` function to generate integer numbers) that are evenly distributed throughout a specified range. The `$rdist_uniform` function is not supported in digital contexts.

```
$rdist_uniform ( seed , start , end [,string]) ;  
$dist_uniform ( seed , start , end [,string]) ;
```

*seed* is a scalar integer variable used to initialize the sequence of generated numbers. *seed* must be a variable because the function updates the value of *seed* at each iteration. To ensure generation of a uniform distribution, change the value of *seed* only when you initialize the sequence.

*start* is an integer or real expression that specifies the smallest number that the `$dist_uniform` function is allowed to return. *start* must be smaller than *end*.

*end* is an integer or real expression that specifies the largest number that the `$dist_uniform` function is allowed to return. *end* must be larger than *start*.

*string* is an optional argument and provides support for Monte Carlo analysis. If *string* is set to `global`, then one value is generated for each Monte Carlo trial. If *string* is set to

## Cadence Verilog-A Language Reference

### Simulator Functions

---

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.

The following module returns a series of real numbers, each of which is between 20 and 60 inclusively.

```
module distcheck (pinout) ;
electrical pinout ;
parameter integer start_range = 20 ;           // A parameter
integer seed, end_range;
real rrandnum ;
analog begin
    @ (initial_step) begin
        seed = 23 ;                           // Initialize the seed just once
        end_range = 60 ;                       // A variable
    end
    rrandnum = $rdist_uniform(seed, start_range, end_range);
    $display ("Random number is %g", rrandnum ) ;
// The next line shows how the seed changes at each
// iterative use of the distribution function.
    $display ("Current seed is %d", seed) ;
    V(pinout) <+ rrandnum ;
end // of analog block
endmodule
```

## Normal (Gaussian) Distribution

Use the `$rdist_normal` function to generate random real numbers (or the `$dist_normal` function to generate integer numbers) that are normally distributed. The `$rdist_normal` function is not supported in digital contexts.

```
$rdist_normal ( seed , mean , standard_deviation [,string] ) ;
$dist_normal ( seed , mean , standard_deviation [,string] ) ;
```

*seed* is a scalar integer variable used to initialize the sequence of generated numbers. *seed* must be a variable because the function updates the value of *seed* at each iteration. To ensure generation of a normal distribution, change the value of *seed* only when you initialize the sequence.

*mean* is an integer or real expression that specifies the value to be approached by the mean value of the generated numbers.

*standard\_deviation* is an integer or real expression that determines the width of spread of the generated values around *mean*. Using a larger *standard\_deviation* spreads the generated values over a wider range.

*string* is an optional argument and provides support for Monte Carlo analysis. If *string* is set to `global`, then one value is generated for each Monte Carlo trial. If *string* is set to

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.

To generate a gaussian distribution, use a *mean* of 0 and a *standard\_deviation* of 1. For example, the following module returns a series of real numbers that together form a gaussian distribution.

```
module distcheck (pinout) ;
electrical pinout ;
integer seed ;
real rrandnum ;

analog begin
  @ (initial_step) begin
    seed = 23 ;
  end
  rrandnum = $rdist_normal( seed, 0, 1 ) ;
  $display ("Random number is %g", rrandnum ) ;
  V(pinout) <+ rrandnum ;
end // of analog block
endmodule
```

## Exponential Distribution

Use the `$rdist_exponential` function to generate random real numbers (or the `$dist_exponential` function to generate integer numbers) that are exponentially distributed. The `$rdist_exponential` function is not supported in digital contexts.

```
$rdist_exponential ( seed , mean [,string] ) ;
$dist_exponential ( seed , mean [,string] ) ;
```

*seed* is a scalar integer variable used to initialize the sequence of generated numbers. *seed* must be a variable because the function updates the value of *seed* at each iteration. To ensure generation of an exponential distribution, change the value of *seed* only when you initialize the sequence.

*mean* is an integer or real value greater than zero. *mean* specifies the value to be approached by the mean value of the generated numbers.

*string* is an optional argument and provides support for Monte Carlo analysis. If *string* is set to `global`, then one value is generated for each Monte Carlo trial. If *string* is set to `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.

For example, the following module returns a series of real numbers that together form an exponential distribution.

```
module distcheck (pinout) ;
electrical pinout ;
integer seed, mean ;
real rrandnum ;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
analog begin
  @ (initial_step) begin
    seed = 23 ;
    mean = 5 ; // Mean must be > 0
  end
  rrandnum = $rdist_exponential(seed, mean) ;
  $display ("Random number is %g", rrandnum) ;
  V(pinout) <+ rrandnum ;
end // of analog block
endmodule
```

## Poisson Distribution

Use the `$rdist_poisson` function to generate random real numbers (or the `$dist_poisson` function to generate integer numbers) that form a Poisson distribution. The `$rdist_poisson` function is not supported in digital contexts.

```
$rdist_poisson ( seed , mean [,string] ) ;
$dist_poisson ( seed , mean [,string] ) ;
```

*seed* is a scalar integer variable used to initialize the sequence of generated numbers. *seed* must be a variable because the function updates the value of *seed* at each iteration. To ensure generation of a Poisson distribution, change the value of *seed* only when you initialize the sequence.

*mean* is an integer or real value greater than zero. *mean* specifies the value to be approached by the mean value of the generated numbers.

*string* is an optional argument and provides support for Monte Carlo analysis. If *string* is set to `global`, then one value is generated for each Monte Carlo trial. If *string* is set to `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.

For example, the following module returns a series of real numbers that together form a Poisson distribution.

```
module distcheck (pinout) ;
  electrical pinout ;
  integer seed, mean ;
  real rrandnum ;
  analog begin
    @ (initial_step) begin
      seed = 23 ;
      mean = 5 ; // Mean must be > 0
    end
    rrandnum = $rdist_poisson(seed, mean) ;
    $display ("Random number is %g", rrandnum) ;
    V(pinout) <+ rrandnum ;
  end // of analog block
endmodule
```



## Chi-Square Distribution

Use the `$rdist_chi_square` function to generate random real numbers (or the `$dist_chi_square` function to generate integer numbers) that form a chi-square distribution. The `$rdist_chi_square` function is not supported in digital contexts.

```
$rdist_chi_square ( seed , degree_of_freedom [,string] ) ;  
$dist_chi_square ( seed , degree_of_freedom [,string] ) ;
```

*seed* is a scalar integer variable used to initialize the sequence of generated numbers. *seed* must be a variable because the function updates the value of *seed* at each iteration. To ensure generation of a chi-square distribution, change the value of *seed* only when you initialize the sequence.

*degree\_of\_freedom* is an integer value greater than zero. *degree\_of\_freedom* determines the width of spread of the generated values. Using a larger *degree\_of\_freedom* spreads the generated values over a wider range.

*string* is an optional argument and provides support for Monte Carlo analysis. If *string* is set to `global`, then one value is generated for each Monte Carlo trial. If *string* is set to `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.

For example, the following module returns a series of real numbers that together form a chi-square distribution.

```
module distcheck (pinout) ;  
  electrical pinout ;  
  integer seed, dof ;  
  real rrandnum ;  
  
  analog begin  
    @ (initial_step) begin  
      seed = 23 ;  
      dof = 5 ;           // Degree of freedom must be > 0  
    end  
    rrandnum = $rdist_chi_square(seed, dof) ;  
    $display ("Random number is %g", rrandnum) ;  
    V(pinout) <+ rrandnum ;  
  end // of analog block  
endmodule
```

## Student's T Distribution

Use the `$rdist_t` function to generate random real numbers (or the `$dist_t` function to generate integer numbers) that form a Student's T distribution. The `$rdist_t` function is not supported in digital contexts.

```
$rdist_t ( seed , degree_of_freedom [,string] ) ;  
$dist_t ( seed , degree_of_freedom [,string] ) ;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

*seed* is a scalar integer variable used to initialize the sequence of generated numbers. *seed* must be a variable because the function updates the value of *seed* at each iteration. To ensure generation of a Student's T distribution, change the value of *seed* only when you initialize the sequence.

*degree\_of\_freedom* is an integer value greater than zero. *degree\_of\_freedom* determines the width of spread of the generated values. Using a larger *degree\_of\_freedom* spreads the generated values over a wider range.

*string* is an optional argument and provides support for Monte Carlo analysis. If *string* is set to *global*, then one value is generated for each Monte Carlo trial. If *string* is set to *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.

For example, the following module returns a series of real numbers that together form a Student's T distribution.

```
module distcheck (pinout) ;
electrical pinout ;
integer seed, dof ;
real rrandnum ;

analog begin
    @ (initial_step) begin
        seed = 23 ;
        dof = 15 ; // Degree of freedom must be > 0
    end
    rrandnum = $rdist_t(seed, dof) ;
    $display ("Random number is %g", rrandnum) ;
    V(pinout) <+ rrandnum ;
end // of analog block

endmodule
```

## Erlang Distribution

Use the `$rdist_erlang` function to generate random real numbers (or the `$dist_erlang` function to generate integer numbers) that form an Erlang distribution. The `$rdist_erlang` function is not supported in digital contexts.

```
$rdist_erlang ( seed , k , mean [,string] ) ;
$dist_erlang ( seed , k , mean [,string] ) ;
```

*seed* is a scalar integer variable used to initialize the sequence of generated numbers. *seed* must be a variable because the function updates the value of *seed* at each iteration. To ensure generation of an Erlang distribution, change the value of *seed* only when you initialize the sequence.

*k* is an integer value greater than zero. Using a larger value for *k* decreases the variance of the distribution.

## Cadence Verilog-A Language Reference

### Simulator Functions

---

*mean* is an integer or real value greater than zero. *mean* specifies the value to be approached by the mean value of the generated numbers.

*string* is an optional argument and provides support for Monte Carlo analysis. If *string* is set to *global*, then one value is generated for each Monte Carlo trial. If *string* is set to *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.

For example, the following module returns a series of real numbers that together form an Erlang distribution.

```
module distcheck (pinout) ;
electrical pinout ;
integer seed, k, mean ;
real rrandnum ;
analog begin
  @ (initial_step) begin
    seed = 23 ;
    k = 20 ;           // k must be > 0
    mean = 15 ;       // Mean must be > 0
  end
  rrandnum = $rdist_erlang(seed, k, mean) ;
  $display ("Random number is %g", rrandnum) ;
  V(pinout) <+ rrandnum ;
end // of analog block
endmodule
```

## Interpolating with Table Models

The various interpolation schemes are lookup, linear, quadratic splines, and cubic splines. The extrapolation may be specified as being constant, linear, or error (meaning if extrapolation occurs the system should error out).

Use the `$table_model` function to model the behavior of a design by interpolating between and extrapolating outside of data points.

```

table_model_declaration ::=
    $table_model(variables , table_source [ , ctrl_string ] )
variables ::=
    independent_var { , independent_var }
table_source ::=
    data_file
    | table_model_array
data_file ::=
    "filename"
    | string_param
table_model_array ::=
    array_ID { , array_ID } , output_array_ID
ctrl_string ::=
    "sub_ctrl_string { , sub_ctrl_string } [ ; dependent_selector ]"
sub_ctrl_string ::=
    I
    | D
    | [ degree_char ] [ extrap_char [ extrap_char ] ]
degree_char ::=
    1 | 2 | 3
extrap_char ::=
    C | L | S | E
dependent_selector ::=
    integer
    
```

*independent\_var* is a numerical expression used as an independent model variable. It can be any legal expression you can assign to an analog signal. You must specify an independent model variable for each dimension with a corresponding *sub\_ctrl\_string* other than I (ignore). You must not specify an independent model variable for dimensions that have a *sub\_ctrl\_string* of I (ignore).

**Note:** The I (ignore) *sub\_ctrl\_string* and support for more than one dimension are extensions beyond the Verilog-AMS LRM, Version 2.2.

*data\_file* is the text file that stores the sample points. You can either specify the file name directly or use a string parameter. For more information, see [“Table Model File Format”](#) on page 158.

*table\_model\_array* is a set of one-dimensional arrays that contains the data points to pass to the `$table_model` function. The size of the arrays is the same as the number of

## Cadence Verilog-A Language Reference

### Simulator Functions

---

sample points. The data is stored in the arrays so that for the  $k^{\text{th}}$  dimension of the  $i^{\text{th}}$  sample point,  $kth\_dim\_array\_identifier[i] = X_{ik}$  and so that for the  $i^{\text{th}}$  sample point  $output\_array\_identifier[i] = Y_i$ . For an example, see [“Example: Preparing Data in One-Dimensional Array Format”](#) on page 160.

`ctrl_string` controls the numerical aspects of the interpolation process. It consists of subcontrol strings for each dimension. 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. 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 through 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. The remaining character(s) in the sub-string specify the extrapolation behavior.

`sub_ctrl_string` specifies the handling for each dimension.

When you specify `I` (ignore), the software ignores the corresponding dimension (column) in the data file. You might use this setting to skip over index numbers, for example. When you associate the `I` (ignore) value with a dimension, you must not specify a corresponding `independent_var` for that dimension.

When you specify `D` (discrete), the software does not use interpolation for this dimension. If the software cannot find the exact value for the dimension in the corresponding dimension in the data file, it issues an error message and the simulation stops.

`degree_char` is the degree of the splines used for interpolation. The degree can be 1 (linear interpolation), 2 (quadratic spline interpolation), and 3 (cubic spline interpolation). The default value is 1.

`extrap_char` controls how the simulator evaluates a point that is outside the region of sample points included in the data file. The `C` (constant) method returns the table endpoint value. The `L` (linear) extrapolation method, which is the default method, models the extrapolation through a tangent line at the end point. 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 `E` (error) extrapolation method. The `E` (error) extrapolation method issues a warning if the `$table_model` function is requested to evaluate a point beyond the interpolation area.

You can specify the extrapolation method to be used for each end of the sample point region. When you do not specify an `extrap_char` value, the linear extrapolation method is used for both ends. When you specify only one `extrap_char` value, the specified extrapolation method is used for both ends. When you specify two `extrap_char` values, the first character specifies the extrapolation method for the end with the smaller coordinate value, and the second character specifies the method for the end with the larger coordinate value.

The `$table_model` function is subject to the same restrictions as analog operators with respect to where the function can be used. For more information, see [“Restrictions on Using Analog Operators”](#) on page 161.

## Table Model File Format

The data in the table model file must be in the form of a family of ordered isolines. An *isoline* is a curve of at least two values generated when one variable is swept and all other variables are held constant. An *ordered isoline* is an isoline in which the sweeping variable is either monotonically increasing or monotonically decreasing. A *monotonically increasing* variable is one in which every subsequent value is equal to or greater than the previous value. A *monotonically decreasing* variable is one in which every subsequent value is equal to or less than the previous value.

For example, a bipolar transistor can be described by a family of isolines, where each isoline is generated by holding the base current constant and sweeping the collector voltage from 0 to some maximum voltage. If the collector voltage sweeps monotonically, the generated isoline is an ordered isoline. In this example, the collector voltage takes many values for each of the isolines so the voltage is the *fastest changing* independent variable and the base current is the *slowest changing* independent variable. You need to know the fastest changing and slowest changing independent variables to arrange the data correctly in the table model file.

The sample points are stored in the file in the following format:

$P_1$   
 $P_2$   
 $P_3$   
...  
 $P_M$

where  $P_i$  ( $i = 1 \dots M$ ) are the sample points. Each sample point  $P_i$  is on a separate line and is represented as a sequence of numbers,  $X_{i1} X_{i2} \dots X_{iN} Y_i$  where  $N$  is the highest dimension of the model,  $X_{ik}$  is the coordinate of the sample point in the  $k$ th dimension, and  $Y_i$  is the model value at this point.  $X_{i1}$  (the leftmost variable) must be the slowest changing variable,  $X_{iN}$  (the rightmost variable other than the model value) must be the fastest changing

## Cadence Verilog-A Language Reference

### Simulator Functions

---

variable, and the other variables must be arranged in between from slowest changing to fastest changing. Comments, which begin with #, can be inserted anywhere in the file and continue to the end of the line.

For example, to create a table model with three ordered isolines representing the function

$$z = f(x, y) = x + y^2$$

you build the model as follows, assuming that you want to have four sample values on each isoline. The  $y$  values used here are all the same and equally spaced on each isoline, but they do not have to be.

Isoline 1:  $x=1$

```
y = 1, 2, 3, 4
z = 2, 5, 10, 17
```

Isoline 2:  $x=2$

```
y = 1, 2, 3, 4
z = 3, 6, 11, 18
```

Isoline 3:  $x=3$

```
y = 1, 2, 3, 4
z = 4, 7, 12, 19
```

Finally, you decide to prefix each row with an index. The function will be specified so as to ignore this new column of data.

You enter the table model data into the file as

```
# Indx is the index column to be ignored.
# x is the slowest changing independent variable.
# y is the fastest changing independent variable.
# z is the table model value at each point.
# Indx  x  y  z
1      1  1  2
2      1  2  5
3      1  3 10
4      1  4 17
5      2  1  3
6      2  2  6
7      2  3 11
8      2  4 18
9      3  1  4
10     3  2  7
11     3  3 12
12     3  4 19
```

### Example: Using the \$table\_model Function

For example, assume that you have a data file named `nmos.tbl`, which contains the data given above. You might use it in a module as follows.

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
\include "disciplines.vams"
\include "constants.vams"

module mynmos (g, d, s);
electrical g, d, s;
inout g, d, s;

analog begin
    I(d, s) <+ $stable_model (V(g, s), V(d, s), "nmos.tbl", "I,3CL,3CL");
end

endmodule
```

In this example, the first column of data is ignored. The independent variables are  $V(g, s)$  and  $V(d, s)$ . The degree of the splines used for interpolation is 3 for each of the two active dimensions. For each of these dimensions, the extrapolation method for the lower end is clamping and the extrapolation for the upper end is linear.

### Example: Preparing Data in One-Dimensional Array Format

In this example, there are 18 sample points. Consequently, each of the one-dimensional arrays contains 18 bits. Each point has two independent variables, represented by  $x$  and  $y$ , and a value, represented by  $f_{xy}$ .

```
module measured_resistance (a, b);
electrical a, b;
inout a, b;
real x[0:17], y[0:17], f_xy[0:17];
analog begin
    @(initial_step) begin
        x[0]= -10; y[0]= -10; f_xy[0]=0; // 0th sample point
        x[1]= -10; y[1]= -8; f_xy[1]= -0.4; // 1st sample point
        x[2]= -10; y[2]= -6; f_xy[2]= -0.8; // 2nd sample point
        x[3]= -9; y[3]= -10; f_xy[3]=0.2;
        x[4]= -9; y[4]= -8; f_xy[4]= -0.2;
        x[5]= -9; y[5]= -6; f_xy[5]= -0.6;
        x[6]= -9; y[6]= -4; f_xy[6]= -1;
        x[7]= -8; y[7]= -10; f_xy[7]=0.4;
        x[8]= -8; y[8]= -9; f_xy[8]=0.2;
        x[9]= -8; y[9]= -7; f_xy[9]= -0.2;
        x[10]= -8; y[10]= -5; f_xy[10]= -0.6;
        x[11]= -8; y[11]= -3; f_xy[11]= -1;
        x[12]= -7; y[12]= -10; f_xy[12]=0.6;
        x[13]= -7; y[13]= -9; f_xy[13]=0.4;
        x[14]= -7; y[14]= -8; f_xy[14]=0.2;
        x[15]= -7; y[15]= -7; f_xy[15]=0;
        x[16]= -7; y[16]= -6; f_xy[16]= -0.2;
        x[17]= -7; y[17]= -5; f_xy[17]= -0.4;
    end
    I(a, b) <+ $stable_model (V(a), V(b), x, y, f_xy, "3L,1L");
end

endmodule
```



## Analog Operators

Analog operators are functions that operate on more than just the current value of their arguments. These functions maintain an internal state and produce a return value that is a function of an input expression, the arguments, and their internal state.

The analog operators are the

- Limited exponential function
- Time derivative operator
- Time integral operator
- Circular integrator operator
- Delay operator
- Transition filter
- Slew filter
- Laplace transform filters
- Z-transform filters

### Restrictions on Using Analog Operators

Analog operators are subject to these restrictions:

- You can use analog operators inside an `if` or `case` construct only if the controlling conditional expression consists entirely of `genvar` expressions, literal numerical constants, parameters, or the `analysis` function.
- You cannot use analog operators in `repeat`, `while`, or `for` statements.
- You cannot use analog operators inside a function.
- You cannot specify a null argument in the argument list of an analog operator.

### Limited Exponential Function

Use the limited exponential function to calculate the exponential of a real argument.

```
limexp( expr )
```

*expr* is a dynamic expression of type `real`.

The `limexp` function limits the iteration step size to improve convergence. `limexp` behaves like the `exp` function, except that using `limexp` to model semiconductor junctions generally results in dramatically improved convergence. For information on the `exp` function, see [“Standard Mathematical Functions”](#) on page 112.

The `limexp` function is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

## Time Derivative Operator

Use the time derivative operator to calculate the time derivative of an argument.

```
ddt( input [ , abstol | nature ] )
```

*input* is a dynamic expression.

*abstol* is a constant specifying the absolute tolerance that applies to the output of the `ddt` operator. Set *abstol* at the largest signal level that you consider negligible.

*nature* is a nature from which the absolute tolerance is to be derived.

The time derivative operator is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

In DC analyses, the `ddt` operator returns 0. In small-signal analyses, the `ddt` operator phase-shifts *expr* according to the following formula.

$$\text{output}(\omega) = j \cdot \omega \cdot \text{input}(\omega)$$

To define a higher order derivative, you must use an internal node or signal. For example, a statement such as the following is illegal.

```
V(out) <+ ddt(ddt(V(in))) // ILLEGAL!
```

For an example illustrating how to define higher order derivatives correctly, see [“Using Integration and Differentiation with Analog Signals”](#) on page 42.

**Note:** You cannot output the result of the `ddt` operator using statements such as `$print`, `$strobe`, and `$fopen`. Instead, you can use an internal node to record the value, then output the value of the internal node.

## Time Integral Operator

Use the time integral operator to calculate the time integral of an argument.

```
idt( input [ , ic [ , assert [ , abstol | nature ] ] ] )
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

*input* is a dynamic expression to be integrated.

*ic* is a dynamic expression specifying the initial condition.

*assert* is a dynamic integer-valued parameter. To reset the integration, set *assert* to a nonzero value.

*abstol* is a constant explicit absolute tolerance that applies to the input of the `idt` operator. Set *abstol* at the largest signal level that you consider negligible.

*nature* is a nature from which the absolute tolerance is to be derived.

The time integral operator is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

The value returned by the `idt` operator during DC or AC analysis depends on which of the parameters you specify.

If you specify	Then <code>idt</code> returns
<i>input</i>	$\int_0^t x(\tau) d\tau$ <p>The time-integral of <i>x</i> from 0 to <i>t</i> with the initial condition being computed in the DC analysis.</p>
<i>input, ic</i>	$\int_0^t x(\tau) d\tau + ic$ <p>The time-integral of <i>x</i> from 0 to <i>t</i> with initial condition <i>ic</i>. In DC or IC analyses, returns <i>ic</i>.</p>
<i>input, ic, assert</i>	$\int_{t_0}^t x(\tau) d\tau + ic$ <p>The time-integral of <i>x</i> from <i>t</i><sub>0</sub> to <i>t</i> with initial condition <i>ic</i>. In DC or IC analyses, and when <i>assert</i> is nonzero, returns <i>ic</i>. <i>t</i><sub>0</sub> is the time when <i>assert</i> last became 0.</p>
<i>input, ic, assert, abstol</i>	$\int_{t_0}^t x(\tau) d\tau + ic$ <p>The time-integral of <i>x</i> from <i>t</i><sub>0</sub> to <i>t</i> with initial condition <i>ic</i>. In DC or IC analysis, and when <i>assert</i> is nonzero, returns <i>ic</i>. <i>t</i><sub>0</sub> is the time when <i>assert</i> last became 0.</p>

If you specify	Then <code>idt</code> returns
<code>input, ic,</code> <code>assert, nature</code>	$\int_{t_0}^t x(\tau) d\tau + ic$ <p>The time-integral of <math>x</math> from <math>t_0</math> to <math>t</math> with initial condition <math>ic</math>. In DC or IC analysis, and when <code>assert</code> is nonzero, returns <math>ic</math>. <math>t_0</math> is the time when <code>assert</code> last became 0.</p>

The initial condition forces the DC solution to the system. You must specify the initial condition, `ic`, unless you are using the `idt` operator in a system with feedback that forces `input` to zero. If you use a model in a feedback configuration, you can leave out the initial condition 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 that forced DC solution.

The following statement illustrates using `idt` with a specified initial condition.

```
V(out) <+ sin(2*`M_PI*(fc*$abstime + idt(gain*V(in),0))) ;
```

## Circular Integrator Operator

Use the circular integrator operator to convert an expression argument into its indefinitely integrated form.

```
idtmod(expr [ , ic [ , modulus [ , offset [ , abstol | nature ] ] ] )
```

`expr` is the dynamic integrand or expression to be integrated.

`ic` is a dynamic initial condition. By default, the value of `ic` is zero.

`modulus` is a dynamic value at which the output of `idtmod` is reset. `modulus` must be a positive value equation. If you do not specify `modulus`, `idtmod` behaves like the `idt` operator and performs no limiting on the output of the integrator.

`offset` is a dynamic value added to the integration. The default is zero.

The `modulus` and `offset` parameters define the bounds of the integral. The output of the `idtmod` function always remains in the range

$$offset < idtmod\_output < offset + modulus$$

`abstol` is a constant explicit absolute tolerance that applies to the input of the `idtmod` operator. Set `abstol` at the largest signal level that you consider negligible.

`nature` is a nature from which the absolute tolerance is to be derived.

## Cadence Verilog-A Language Reference

### Simulator Functions

---

The circular integrator operator is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

The value returned by the `idtmod` operator depends on which parameters you specify.

If you specify	Then <code>idtmod</code> returns
<code>expr</code>	$x = \int_0^t \text{expr}(\tau) d\tau$ <p>The time-integral of <code>expr</code> from 0 to <code>t</code> with the initial condition being computed in the DC analysis. Returns <code>x</code>.</p>
<code>expr, ic</code>	$x = \int_0^t \text{expr}(\tau) d\tau + ic$ <p>The time-integral of <code>expr</code> from 0 to <code>t</code> with initial condition <code>ic</code>. In DC or IC analysis, returns <code>ic</code>; otherwise, returns <code>x</code>.</p>
<code>expr, ic, modulus</code>	$x = \int_0^t \text{expr}(\tau) d\tau + ic$ <p>where <math>x = n * \text{modulus} + k</math>  <math>n = \dots -3, -2, -1, 0, 1, 2, 3 \dots</math>  Returns <code>k</code> where <math>0 &lt; k &lt; \text{modulus}</math></p>
<code>expr, ic, modulus, offset</code>	$x = \int_0^t \text{expr}(\tau) d\tau + ic$ <p>where <math>x = n * \text{modulus} + k</math>  Returns <code>k</code> where <math>\text{offset} &lt; k &lt; \text{offset} + \text{modulus}</math></p>
<code>expr, ic, modulus, offset, abstol</code>	$x = \int_0^t \text{expr}(\tau) d\tau + ic$ <p>where <math>x = n * \text{modulus} + k</math>  Returns <code>k</code> where <math>\text{offset} &lt; k &lt; \text{offset} + \text{modulus}</math></p>
<code>expr, ic, modulus, offset, nature</code>	$x = \int_0^t \text{expr}(\tau) d\tau + ic$ <p>where <math>x = n * \text{modulus} + k</math>  Returns <code>k</code> where <math>\text{offset} &lt; k &lt; \text{offset} + \text{modulus}</math></p>

The initial condition forces the DC solution to the system. You must specify the initial condition, `ic`, unless you are using `idtmod` in a system with feedback that forces `expr` to

zero. If you use a model in a feedback configuration, you can leave out the initial condition without any unexpected behavior during simulation.

### Example

The circular integrator is useful in cases where the integral can get very large, such as in a voltage controlled oscillator (VCO). For example, you might use the following approach to generate arguments in the range  $[0, 2\pi]$  for the sinusoid.

```
phase = idtmod(fc + gain*V(IN), 0, 1, 0); //Phase is in range [0,1].
V(OUT) <+ sin(2*PI*phase);
```

### Derivative Operator

Use the `ddx` operator to access symbolically-computed partial derivatives of expressions in the analog block.

```
ddx (expr, potential_access_id (net_or_port_scalar_expr))
ddx (expr, flow_access_id (branch_id))
```

*expr* is a real or integer value expression. The derivative operator returns the partial derivative of this argument with respect to the unknown indicated by the second argument, with all other unknowns held constant and evaluated at the current operating point. If *expr* does not depend explicitly on the unknown, the derivative operator returns zero. The *expr* argument:

- Cannot be a dynamic expression, such as `ddx(ddt(...), ...)`
- Cannot be a nested expression, such as `ddx(ddx(...), ...)`
- Cannot include symbolically calculated expressions, such as `ddx(transition(...), ...)`
- Cannot include arrays, such as `ddx(a[0], ...)`
- Cannot contain unknown variables in the system of equations, such as `ddx(V(a), ...)`
- Cannot contain quantities that depend on other quantities, such as:  
`I(a,b)<+g*V(a,b); ddx(I(a,b), V(a))`

*potential\_access\_id* is the access operator for the potential of a scalar net or port.

*net\_or\_port\_scalar\_expr* is a scalar net or port.

*flow\_access\_id* is the access operator for the flow through a branch.

*branch\_id* is the name of a branch.

The derivative operator is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

### Example

This example implements a voltage-controlled dependent current source. The names of the variables indicate the values of the partial derivatives: +1, -1, or 0. These values (scaled by the parameter *k*) can be used in a Newton-Raphson solution.

```
module vccs(pout,nout,pin,nin);
    electrical pout, nout, pin, nin;
    inout 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
```

### Delay Operator

Use the `absdelay` operator to delay the entire signal of a continuously valued waveform.

`absdelay( expr , time_delay [ , max_delay ] )`

*expr* is a dynamic expression to be delayed.

*time\_delay*, a dynamic nonnegative value, is the length of the delay. If you specify *max\_delay*, you can change the value of *time\_delay* during a simulation, as long as the value remains in the range  $0 < time\_delay < max\_delay$ . Typically *time\_delay* is a constant but can also vary with time (when *max\_delay* is defined).

*max\_delay* is a constant nonnegative number greater than or equal to *time\_delay*. You cannot change *max\_delay* because the simulator ignores any attempted changes and continues to use the initial value.

For example, to delay an input voltage you might code

```
V(out) <+ absdelay(V(in), 5u) ;
```

The `absdelay` operator is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

In DC and operating analyses, the `absdelay` operator returns the value of `expr` unchanged. In small-signal analyses, the `absdelay` operator phase-shifts `expr` according to the following formula.

$$output(\omega) = input(\omega) \cdot e^{-j\omega \cdot time\_delay}$$

In time-domain analyses, the `absdelay` operator introduces a transport delay equal to the instantaneous value of `time_delay` based on the following formula.

$$Output(t) = Input(\max(t-time\_delay, 0))$$

## Transition Filter

Use the `transition` filter to smooth piecewise constant waveforms, such as digital logic waveforms. The `transition` filter returns a real number that over time describes a piecewise linear waveform. The `transition` filter also causes the simulator to place time points at both corners of a transition to assure that each transition is adequately resolved.

```
transition(input [, delay [, rise_time [, fall_time [, time_tol ]]])
```

`input` is a dynamic input expression that describes a piecewise constant waveform. It must have a real value. In DC analysis, the `transition` filter simply returns the value of `input`. Changes in `input` do not have an effect on the output value until `delay` seconds have passed.

`delay` is a dynamic nonnegative real value that is an initial delay. By default, `delay` has a value of zero.

`rise_time` is a dynamic positive real value specifying the time over which you want positive transitions to occur. If you do not specify `rise_time` or if you give `rise_time` a value of 0, `rise_time` defaults to the value defined by `'default_transition`.

`fall_time` is a dynamic positive real number specifying the time over which you want negative transitions to occur. By default, `fall_time` has the same value that `rise_time` has. If you do not specify `rise_time` or if you give `rise_time` a value of 0, `fall_time` defaults to the value defined by `'default_transition`.

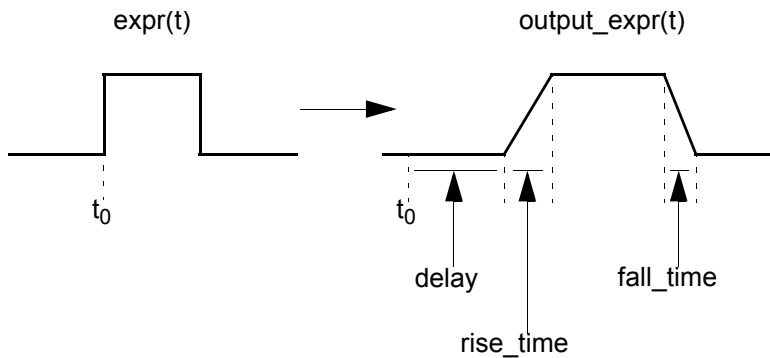
`time_tol` is a constant expression with a positive value. This option requires the simulator to place time points no more than the value of `time_tol` away from the two corners of the transition.

If `'default_transition` is not specified, the default behavior of the `transition` filter approximates the ideal behavior of a zero-duration transition.



The `transition` filter is subject to the restrictions listed in “[Restrictions on Using Analog Operators](#)” on page 161.

With the `transition` filter, you can control transitions between discrete signal levels by setting the rise time and fall time of signal transitions. The `transition` filter stretches instantaneous changes in signals over a finite amount of time, as shown below, and can also delay the transitions.



Use short transitions with caution because they can cause the simulator to slow down to meet accuracy constraints.

The next code fragment demonstrates how the `transition` filter might be used.

```
// comparator model
analog begin
  if ( V(in) > 0 ) begin
    Vout = 5 ;
  end
  else begin
    Vout = 0 ;
  end
  end
  V(out) <+ transition(Vout) ;
end
```

### **Caution**

**The transition filter is designed to smooth out piecewise constant waveforms. If you apply the transition filter to smoothly varying waveforms, the simulator might run slowly, and the results will probably be unsatisfactory. For smoothly varying waveforms, consider using the `slew` filter instead. For information, see “[Slew Filter](#)” on page 171.**

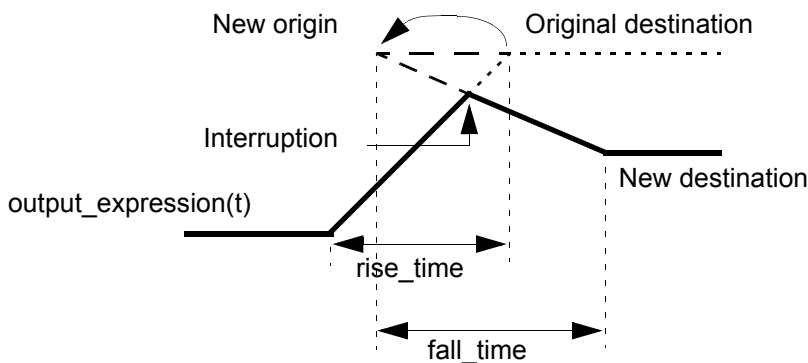
## Cadence Verilog-A Language Reference

### Simulator Functions

If interrupted on a rising transition, the `transition` filter adjusts the slope so that at the revised end of the transition the value is that of the new destination.

<b>If the new destination value is <i>below</i> the value at the point of interruption, the <code>transition</code> filter</b>	<b>If the new destination value is <i>above</i> the value at the point of interruption, the <code>transition</code> filter</b>
<ol style="list-style-type: none"> <li>1. Uses the value of the original destination as the value of the new origin.</li> <li>2. Adjusts the slope of the transition to the rate at which the value would decay from the value of the new origin to the value of the new destination in <code>fall_time</code> seconds.</li> <li>3. Causes the value of the filter output to decay at the new slope, from the value at the point of interruption to the value at the new destination.</li> </ol>	<ol style="list-style-type: none"> <li>1. Retains the original origin.</li> <li>2. Adjusts the slope of the transition to the rate at which the value would increase from the value of the origin to the value of the new destination in <code>rise_time</code> seconds.</li> <li>3. Causes the value of the filter output to increase at the new slope, from the value at the point of interruption to the value at the new destination.</li> </ol>

In the following example, a rising transition is interrupted when it is about three fourths complete, and the value of the new destination is below the value at the point of interruption. The `transition` filter computes the slope that would complete a transition from the new origin (not the value at the point of interruption) in the specified `fall_time`. The `transition` filter then uses the computed slope to transition from the current value to the new destination.



An interruption in a falling transition causes the transition filter to behave in an equivalent manner.

With larger delays, it is possible for a new transition to be specified before a previously specified transition starts. The `transition` filter handles this by deleting any transitions that would follow a newly scheduled transition. A `transition` filter can have an arbitrary number of transitions pending. You can use a `transition` filter in this way to implement the transport delay of discretely valued signals.

The following example implements a D-type flip flop. The `transition` filter smooths the output waveforms.

```
module d_ff(vin_d, vclk, vout_q, vout_qbar) ;
input vclk, vin_d ;
output vout_q, vout_qbar ;
electrical vout_q, vout_qbar, vclk, vin_d ;
parameter real vlogic_high = 5 ;
parameter real vlogic_low = 0 ;
parameter real vtrans_clk = 2.5 ;
parameter real vtrans = 2.5 ;
parameter real tdel = 3u from [0:inf) ;
parameter real trise = 1u from (0:inf) ;
parameter real tfall = 1u from (0:inf) ;
integer x ;
analog begin
    @ (cross( V(vclk) - vtrans_clk, +1 )) x = (V(vin_d) > vtrans) ;
    V(vout_q) <+ transition( vlogic_high*x + vlogic_low!*x, tdel, trise, tfall ) ;
    V(vout_qbar) <+ transition( vlogic_high!*x + vlogic_low*x, tdel,
                                trise, tfall ) ;
end
endmodule
```

The following example illustrates a use of the `transition` filter that should be avoided. The expression is dependent on a continuous signal and, as a consequence, the filter runs slowly.

```
I(p, n) <+ transition(V(p, n)/out1, tdel, trise, tfall); // Do not do this.
```

However, you can use the following approach to implement the same behavior in a statement that runs much faster.

```
I(p, n) <+ V(p, n) * transition(1/out1, tdel, trise, tfall); // Do this instead.
```

## Slew Filter

Use the `slew` filter to control the rate of change of a waveform. A typical use for `slew` is generating continuous signals from piecewise continuous signals. For discrete signals, consider using the `transition` filter instead. See [“Transition Filter”](#) on page 168 for more information.

```
slew(input [ , max_pos_rate [ , max_neg_rate ] ] )
```

*input* is a dynamic expression with a real value. In DC analysis, the `slew` filter simply returns the value of *input*.

## Cadence Verilog-A Language Reference

### Simulator Functions

---

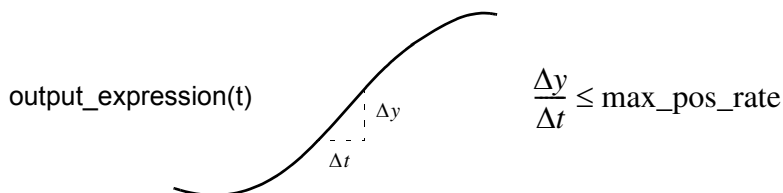
*max\_pos\_rate* is a dynamic real number greater than zero, which is the maximum positive slew rate.

*max\_neg\_rate* is a dynamic real number less than zero, which is the maximum negative slew rate.

If you specify only one rate, its absolute value is used for both rates. If you give no rates, *slew* passes the signal through unchanged. If the rate of change of *input* is less than the specified maximum slew rates, *slew* returns the value of *input*.

The *slew* filter is subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

When applied, *slew* forces all transitions of *expr* faster than *max\_pos\_rate* to change at the *max\_pos\_rate* rate for positive transitions and limits negative transitions to the *max\_neg\_rate* rate.



The *slew* filter is particularly valuable for controlling the rate of change of sinusoidal waveforms. The *transition* function distorts such signals, whereas *slew* preserves the general shape of the waveform. The following 4-bit digital-to-analog converter uses the *slew* function to control the rate of change of the analog signal at its output.

```
module dac4(d, out) ;
input [0:3] d ;
inout out ;
electrical [0:3] d ;
electrical out ;
parameter real slewrate = 0.1e6 from (0:inf) ;

    real Ti ;
    real Vref ;
    real scale_fact ;

    analog begin
        Ti = 0 ;
        Vref = 1.0 ;
        scale_fact = 2 ;
        generate ii (3,0,-1) begin
            Ti = Ti + ((V(d[ii]) > 2.5) ? (1.0/scale_fact) : 0);
            scale_fact = scale_fact/2 ;
        end
        V(out) <+ slew( Ti*Vref, slewrate ) ;
    end
endmodule
```

## Implementing Laplace Transform S-Domain Filters

The Laplace transform filters implement lumped linear continuous-time filters. Each filter accepts an optional absolute tolerance parameter  $\epsilon$ , which this release of Verilog-A ignores. The set of array values that are used to define the poles and zeros, or numerator and denominator, of a filter the first time it is used during an analysis are used at all subsequent time points of the analysis. As a result, changing array values during an analysis has no effect on the filter.

The Laplace transform filters are subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161. However, while most analog functions can be used, with certain restrictions, in `if` or `case` constructs, the Laplace transform filters cannot be used in `if` or `case` constructs in any circumstances.

### **Arguments Represented as Vectors**

If you use an argument represented as a vector to define a numerator in a Laplace filter, and if one or more of the elements in the vector are 0, the order of the numerator is determined by the *position* of the rightmost non-zero vector element. For example, in the following module, the order of the numerator, `nn`, is 1

```
module test(pin, nin, pout, nout);
  electrical pin, nin, pout, nout;
  real nn[0:2];
  real dd[0:2];
  analog begin
    @(initial_step) begin
      nn[0] = 1; // The highest order non-zero coefficient of the numerator.
      nn[1] = 0;
      nn[2] = 0;
      dd[0] = 1;
      dd[1] = 1;
      dd[2] = 1;
    end
    V(pout, nout) <+ laplace_nd(V(pin,nin), nn, dd);
  end
endmodule
```

### **Arguments Represented as Arrays**

If you use an argument represented as an array constant to define a numerator in a Laplace filter, and if one or more of the elements in the array constant are 0, the order of the numerator is determined by the *position* of the rightmost non-zero array element. For example, if your numerator array constant is `{1,0,0}`, the order of the numerator is 1. If your array constant is `{1,0,1}`, the order of the numerator is 3. In the following example, the numerator order is 1 (and the value is 1).

```
module test(pin, nin, pout, nout);
  electrical pin, nin, pout, nout;
  analog begin
    V(pout, nout) <+ laplace_nd(V(pin,nin), {1,0,0}, {1,1,1});
  end
endmodule
```

Array literals used for the Laplace transforms can also take the form that uses a back tic. For example,

```
V(out) <+ laplace_nd(`{5,6}, `{7.8,9.0});
```

### Zero-Pole Laplace Transforms

Use `laplace_zp` to implement the zero-pole form of the Laplace transform filter.

```
laplace_zp(expr,  $\zeta$ ,  $\rho$  [,  $\varepsilon$ ])
```

$\zeta$  (zeta) is a fixed-sized 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.  $\rho$  (rho) is a fixed-sized vector of N real pairs, one for each pole. Specify the poles in the same manner as the zeros. If you use array literals to define the  $\zeta$  and  $\rho$  vectors, the values must be constant or dependent upon parameters only. You cannot use array literal values defined by variables.

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  $\zeta_k^r$  and  $\zeta_k^i$  are the real and imaginary parts of the  $k^{th}$  zero, and  $\rho_k^r$  and  $\rho_k^i$  are the real and imaginary parts of the  $k^{th}$  pole.

If a root (a pole or zero) is real, you must specify the imaginary part as 0. If a root is complex, its conjugate must be present. If a root is zero, the term associated with it is implemented as  $s$  rather than  $(1 - s/r)$ , where  $r$  is the root. If the list of roots is empty, unity is used for the corresponding denominator or numerator.

## Zero-Denominator Laplace Transforms

Use `laplace_zd` to implement the zero-denominator form of the Laplace transform filter.

`laplace_zd(expr, ζ, d[, ε])`

$\zeta$  (zeta) is a fixed-sized 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.  $d$  is a fixed-sized vector of N real numbers that contains the coefficients of the denominator. If you use array literals to define the  $\zeta$  and  $d$  vectors, the values must be constant or dependent upon parameters only. You cannot use array literal values defined by variables.

The transfer function is

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

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

## Numerator-Pole Laplace Transforms

Use `laplace_np` to implement the numerator-pole form of the Laplace transform filter.

`laplace_np(expr, n, ρ[, ε])`

$n$  is a fixed-sized vector of M real numbers that contains the coefficients of the numerator.  $\rho$  (rho) is a fixed-sized 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. If you use array literals to define the  $n$  and  $\rho$  vectors, the array values must be constant or dependent upon parameters only. You cannot use array values defined by variables.

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  $k^{\text{th}}$  power of  $s$  in the numerator, and  $\rho_k^r$  and  $\rho_k^i$  are the real and imaginary parts of the  $k^{\text{th}}$  pole. If a pole is real, you must specify the imaginary part as 0. If a pole is complex, its conjugate must be present. If a pole is zero, the term associated with it is implemented as  $s$  rather than  $(1 - s/\rho)$ .

### Numerator-Denominator Laplace Transforms

Use `laplace_nd` to implement the numerator-denominator form of the Laplace transform filter.

`laplace_nd(expr, n, d[, ε])`

$n$  is a fixed-sized vector of  $M$  real numbers that contains the coefficients of the numerator, and  $d$  is a fixed-sized vector of  $N$  real numbers that contains the coefficients of the denominator. If you use array literals to define the  $n$  and  $d$  vectors, the values must be constant or dependent upon parameters only. You cannot use array values defined by variables.

The transfer function is

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

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



## Examples

The following code fragments illustrate how to use the Laplace transform filters.

```
V(out) <+ laplace_zp(V(in), {0,0}, {1,2,1,-2});
```

implements

$$H(s) = \frac{s}{\left(1 - \frac{s}{1+2j}\right)\left(1 - \frac{s}{1-2j}\right)} = \frac{s}{1 - 0.4s + 0.2s^2}$$

The code fragment

```
V(out) <+ laplace_nd(V(in), {0,1}, {1,-0.4,0.2});
```

is equivalent.

The following statement contains an empty vector such that the middle argument is null:

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

The absence of zeros, indicated by the null argument, means that the transfer function reduces to the following equation:

$$H(s) = \frac{1}{1+s}$$

The next module illustrates the use of array literals that depend on parameters. In this code, the array literal `{dx, 6*dx, 5*dx}` depends on the value of the parameter `dx`.

```
module svcvs_zd(pin, nin, pout, nout);
    electrical pin, nin, pout, nout;
    parameter real nx = 0.5;
    parameter integer dx = 1;
    analog begin
        V(pout,nout) <+ laplace_zd(V(pin,nin), {0-nx,0}, {dx,6*dx,5*dx});
    end
endmodule
```

The next fragment illustrates an efficient way to initialize array values. Because only the initial set of array values used by a filter has any effect, this example shows how you can use the `initial_step` event to set values at the beginning of the specified analyses.

```
real nn[0:1] ;
real dd[0:2] ;
analog begin
    @(initial_step("static")) begin
        nn[0] = 1 ;           // These assignment
        nn[1] = 2 ;           // statements run only
        dd[0] = 1 ;           // at the beginning of
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
        dd[1] = 6 ;           // the analyses.
    end
    V(pout, nout) <+ laplace_nd(V(pin,nin), nn, dd) ;
end
```

When you use this technique, be sure to initialize the arrays at the beginning of each analysis that uses the filter. The `static` analysis is the dc operating point calculation required by most analyses, including `tran`, `ac`, and `noise`. Initializing the array during the `static` phase ensures that the array is non-zero as these analyses proceed.

The next modules illustrate how you can use an array variable to avoid error messages about using array literals with variable dependencies in the Laplace filters. The first version causes an error message.

```
// This version does not work.
`include "constants.vams"
`include "disciplines.vams"

module laplace(out, in);
inout in, out;
electrical in, out;
real dummy;

    analog begin
        dummy = -0.5;
        V(out) <+ laplace_zd(V(in), [dummy,0], [1,6,5]); //Illegal!
    end
endmodule
```

The next version works as expected.

```
// This version works correctly.
`include "constants.vams"
`include "disciplines.vams"

module laplace(out, in);
inout in, out;
electrical in, out;
real dummy;

real nn[0:1];

analog begin
    dummy = -0.5;
    @(initial_step) begin // Defines the array variable.
        nn[0] = dummy;
        nn[1] = 0;
    end

    V(out) <+ laplace_zd(V(in), nn, [1,6,5]);
end
endmodule
```

## Implementing Z-Transform Filters

The Z-transform filters implement linear discrete-time filters. Each filter requires you to specify a parameter  $T$ , the sampling period of the filter. A filter with unity transfer function acts like a simple sample-and-hold that samples every  $T$  seconds.

All Z-transform filters share three common arguments,  $T$ ,  $\tau$ , and  $t_0$ . The  $T$  argument specifies the period of the filter and must be positive.  $\tau$  specifies the transition time and must be nonnegative. If you specify a nonzero transition time, the simulator controls the time step to accurately resolve both the leading and trailing corner of the transition. If you do not specify a transition time,  $\tau$  defaults to one unit of time as defined by the ``default_transition` compiler directive. If you specify a transition time of 0, the output is abruptly discontinuous. Avoid assigning a Z-filter with 0 transition time directly to a branch because doing so greatly slows the simulation. Finally,  $t_0$  specifies the time of the first sample/transition and is also optional. If not given, the first transition occurs at  $t=0$ .

The values of  $T$  and  $t_0$  at the first time point in the analysis are stored, and those stored values are used at all subsequent time points. The array values used to define a filter are used at all subsequent time points, so changing array values during an analysis has no effect on the filter.

The Z-transform filters are subject to the restrictions listed in [“Restrictions on Using Analog Operators”](#) on page 161.

In small-signal analyses, the Z-transform filters phase-shift *input* according to the following formula.

$$output(\omega) = H(e^{j\omega T}) \cdot input(\omega)$$

### Zero-Pole Z-Transforms

Use `zi_zp` to implement the zero-pole form of the Z-transform filter.

`zi_zp(expr,  $\zeta$ ,  $\rho$ ,  $T$  [ ,  $\tau$  [ ,  $t_0$  ] ] )`

$\zeta$  (zeta) is a fixed or parameter-sized 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.  $\rho$  (rho) is a fixed or parameter-sized vector of N real pairs, one for each pole. The poles are given in the same manner as the zeros. If you use array literals to define the  $\zeta$  and  $\rho$  vectors, the values must be constant or dependent upon parameters only. You cannot use array values defined by variables.

The transfer function is

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

where  $\zeta_k^r$  and  $\zeta_k^i$  are the real and imaginary parts of the  $k^{\text{th}}$  zero, and  $\rho_k^r$  and  $\rho_k^i$  are the real and imaginary parts of the  $k^{\text{th}}$  pole. If a root (a pole or zero) is real, you must specify the imaginary part as 0. If a root is complex, its conjugate must also be present. If a root is the origin, the term associated with it is implemented as  $z$  rather than  $(1 - (z^{-1} \cdot r))$ , where  $r$  is the root. If a list of poles or zeros is empty, unity is used for the corresponding denominator or numerator.

### Zero-Denominator Z-Transforms

Use `zi_zd` to implement the zero-denominator form of the Z-transform filter.

**zi\_zd**(*expr*,  $\zeta$ , *d*, *T* [,  $\tau$  [, *t<sub>0</sub>* ] ])

$\zeta$  (zeta) is a fixed or parameter-sized 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. *d* is a fixed or parameter-sized vector of N real numbers that contains the coefficients of the denominator. If you use array literals to define the  $\zeta$  and *d* vectors, the values must be constant or dependent upon parameters only. You cannot use array values defined by variables.

The transfer function is

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

where  $\zeta_k^r$  and  $\zeta_k^i$  are the real and imaginary parts of the  $k^{\text{th}}$  zero, and  $d_k$  is the coefficient of the  $k^{\text{th}}$  power of  $z$  in the denominator. If a zero is real, you must specify the imaginary part as 0. If a zero is complex, its conjugate must also be present. If a zero is the origin, the term associated with it is implemented as  $z$  rather than  $(1 - (z^{-1} \cdot \zeta))$ .

### Numerator-Pole Z-Transforms

Use `zi_np` to implement the numerator-pole form of the Z-transform filter.

`zi_np(expr, n, rho, T [ , tau [ , t0] ])`

$n$  is a fixed or parameter-sized vector of  $M$  real numbers that contains the coefficients of the numerator.  $\rho$  (rho) is a fixed or parameter-sized 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. If you use array literals to define the  $n$  and  $\rho$  vectors, the values must be constant or dependent upon parameters only. You cannot use array values defined by variables.

The transfer function is

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

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

### Numerator-Denominator Z-Transforms

Use `zi_nd` to implement the numerator-denominator form of the Z-transform filter.

`zi_nd(expr, n, d, T [ , τ [ , t0 ] ])`

$n$  is a fixed or parameter-sized vector of  $M$  real numbers that contains the coefficients of the numerator, and  $d$  is a fixed or parameter-sized vector of  $N$  real numbers that contains the coefficients of the denominator. If you use array literals to define the  $n$  and  $d$  vectors, the values must be constant or dependent upon parameters only. You cannot use array values defined by variables.

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  $k^{\text{th}}$  power of  $z$  in the numerator, and  $d_k$  is the coefficient of the  $k^{\text{th}}$  power of  $s$  in the denominator.

## Examples

The following example illustrates an ideal sampled data integrator with the transfer function

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

This transfer function can be implemented as

```
module ideal_int (in, out) ;
electrical in, out ;
parameter real T = 0.1m ;
parameter real tt = 0.02n ;
parameter real td = 0.04m ;
analog begin
    // The filter is defined with constant array literals.
    V(out) <+ zi_nd(V(in), {1}, {1,-1}, T, tt, td) ;
end
endmodule
```

The next example illustrates additional ways to use parameters and arrays to define filters.

```
module zi (in, out);
electrical in, out;

parameter real T = 0.1;
parameter real tt = 0.02m;
parameter real td = 0.04m;
parameter real n0 = 1;

parameter integer start_num = 0;
parameter integer num_d = 2;

real nn[0:0]; // Fixed-sized array
real dd[start_num:start_num+num_d-1]; // Parameter-sized array
real d;

analog begin
    // The arrays are initialized at the beginning of the listed analyses.
    @(initial_step("ac","dc","tran")) begin
        d = 1*n0;
        nn[start_num] = n0;
        dd[start_num] = d; dd[1] = -d;
    end
    V(out) <+ zi_nd( V(in), nn, dd, T, tt, td);
end
endmodule
```

## Displaying Results

Verilog-A provides these tasks for displaying information: \$strobe, \$display, \$write, and \$debug.

## **\$strobe**

Use the `$strobe` task to display information on the screen. `$strobe` and `$display` use the same arguments and are completely interchangeable. `$strobe` is supported in both analog and digital contexts.

```
strobe_task ::=  
    $strobe [ ( { list_of_arguments } ) ]  
list_of_arguments ::=  
    argument  
    | list_of_arguments , argument
```

The `$strobe` task prints a new-line character after the final argument. A `$strobe` task without any arguments prints only a new-line character.

Each *argument* is a quoted string or an expression that returns a value.

Each quoted string is a set of ordinary characters, special characters, or conversion specifications, all enclosed in one set of quotation marks. Each conversion specification in the string must have a corresponding argument following the string. You must ensure that the type of each argument is appropriate for the corresponding conversion specification.

You can specify an argument without a corresponding conversion specification. If you do, an integer argument is displayed using the `%d` format, and a real argument is displayed using the `%g` format.

## **Special Characters**

Use the following sequences to include the specified characters and information in a quoted string.

---

<b>Use this sequence</b>	<b>To include</b>
<code>\n</code>	The new-line character
<code>\t</code>	The tab character
<code>\\</code>	The backslash character, <code>\</code>
<code>\"</code>	The quotation mark character, <code>"</code>
<code>\ddd</code>	A character specified by 1 to 3 octal digits
<code>%%</code>	The percent character, <code>%</code>
<code>%m</code> or <code>%M</code>	The hierarchical name of the current module, function, or named block

---



## Cadence Verilog-A Language Reference

### Simulator Functions

---

### Conversion Specifications

Conversion specifications have the form

```
% [ flag ] [ field_width ] [ . precision ] format_character
```

where *flag*, *field\_width*, and *precision* can be used only with a real argument.

*flag* is one of the three choices shown in the table:

---

<b>flag</b>	<b>Meaning</b>
-	Left justify the output
+	Always print a sign
Blank space, or any character other than a sign	Print a space

---

*field\_width* is an integer specifying the minimum width for the field.

*precision* is an integer specifying the number of digits to the right of the decimal point.

*format\_character* is one of the following characters.

---

<b>format_ character</b>	<b>Type of Argument</b>	<b>Output</b>	<b>Example Output</b>
b or B		Binary format	0000000000000000 00000000111000
c or C	Integer	ASCII character format	
d or D	Integer	Decimal format	191, 48, -567
e or E	Real	Real, exponential format	-1.0, 4e8, 34.349e-12
f or F	Real	Real, fixed-point format	191.04, -4.789
g or G	Real	Real, exponential, or decimal format, whichever format results in the shortest printed output	9.6001, 7.34E-8, -23.1E6
h or H	Integer	Hexadecimal format	3e, 262, a38, fff, 3E, A38

## Cadence Verilog-A Language Reference Simulator Functions

---

format_ character	Type of Argument	Output	Example Output
o or O	Integer	Octal format	127, 777
r or R	Real	Engineering notation format	123,457M, 12.345K
s or S	String constant	String format	

---

### Examples of \$strobe Formatting

Assume that module `format_module` is instantiated in a netlist file with the instantiation

```
formatTest format_module
```

The module is defined as

```
module format_module ;
integer ival ;
real rval ;
analog begin
    ival = 98 ;
    rval = 123.456789 ;
    $strobe("Format c gives %c" , ival) ;
    $strobe("Format C gives %C" , ival) ;
    $strobe("Format d gives %d" , ival) ;
    $strobe("Format D gives %D" , ival) ;
    $strobe("Format e (real) gives %e" , rval) ;
    $strobe("Format E (real) gives %E" , rval) ;
    $strobe("Format f (real) gives %f" , rval) ;
    $strobe("Format F (real) gives %F" , rval) ;
    $strobe("Format g (real) gives %g" , rval) ;
    $strobe("Format G (real) gives %G" , rval) ;
    $strobe("Format h gives %h" , ival) ;
    $strobe("Format H gives %H" , ival) ;
    $strobe("Format m gives %m") ;
    $strobe("Format M gives %M") ;
    $strobe("Format o gives %o" , ival) ;
    $strobe("Format O gives %O" , ival) ;
    $strobe("Format r (real) gives %r" , rval*10) ;
    $strobe("Format R (real) gives %R" , rval*10) ;
    $strobe("Format s gives %s" , "s string") ;
    $strobe("Format S gives %S" , "S string") ;
    $strobe("newline,\ntab,\tback-slash, \\") ;
    $strobe("doublequote,\"") ;
end
endmodule
```

When you run `format_module`, it displays

```
Format c gives b
Format C gives b
Format d gives 98
Format D gives 98
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
Format e gives 1.234568e+02
Format E gives 1.234568e+02
Format f gives 123.456789
Format F gives 123.456789
Format g gives 123.457
Format G gives 123.457
Format h gives 62
Format H gives 62
Format m gives formatTest
Format M gives formatTest
Format o gives 142
Format O gives 142
Format r gives 1.23K
Format R gives 1.23K
Format s gives s string
Format S gives S string
newline,
tab,      back-slash, \
doublequote,"
```

### **\$display**

Use the `$display` task to display information on the screen. `$display` is supported in both analog and digital contexts.

```
display_task ::=
    $display [ ( { list_of_arguments } ) ]

list_of_arguments ::=
    argument
    | list_of_arguments , argument
```

`$display` and `$strobe` use the same arguments and are completely interchangeable. For guidance, see “[\\$strobe](#)” on page 184.

### **\$write**

Use the `$write` task to display information on the screen. This task is identical to the `$strobe` task, except that `$strobe` automatically adds a newline character to the end of its output, whereas `$write` does not. `$write` is supported in both analog and digital contexts.

```
write_task ::=
    $write [ ( { list_of_arguments } ) ]

list_of_arguments ::=
    argument
    | list_of_arguments , argument
```

The arguments you can use in `list_of_arguments` are the same as those used for `$strobe`. For guidance, see “[\\$strobe](#)” on page 184.

## \$debug

Use the `$debug` task to display information on the screen while the analog solver is running. This task displays the values of the arguments for each iteration of the solver.

```
debug_task ::=
    $debug [ ( { list_of_arguments } ) ]
list_of_arguments ::=
    argument
    | list_of_arguments , argument
```

The arguments you can use in `list_of_arguments` are the same as those used for `$strobe`. For guidance, see “[\\$strobe](#)” on page 184.

## Specifying Power Consumption

Use the `$pwr` system task to specify the power consumption of a module. The `$pwr` task is supported in only analog contexts.

**Note:** The `$pwr` task is a nonstandard Cadence-specific language extension.

```
pwr_task ::=
    $pwr( expression )
```

*expression* is an expression that specifies the power contribution. If you specify more than one `$pwr` task in a behavioral description, the result of the `$pwr` task is the sum of the individual contributions.

To ensure a useful result, your module must contain an assignment inside the behavior specification. Your module must also compute the value of `$pwr` tasks at every iteration. If these conditions are not met, the result of the `$pwr` task is zero.

The `$pwr` task does not return a value and cannot be used inside other expressions. Instead, access the result by using the `options` and `save` statements in the netlist. For example, using the following statement in the netlist saves all the individual power contributions and the sum of the contributions in the module named *name*:

```
name options pwr=all
```

For `save`, use a statement like the following:

```
save name:pwr
```

In each format, *name* is the name of a module.

For more information about the `options` statement, see [Chapter 7](#) of the *Spectre Circuit Simulator User Guide*. For more about the `save` statement, see [Chapter 8](#) of the *Spectre Circuit Simulator User Guide*.

## Example

```
// Resistor with power contribution
`include "disciplines.vams"

module Res(pos, neg);
  inout pos, neg;
  electrical pos, neg;
  parameter real r=5;
  analog begin
    V(pos,neg) <+ r * I(pos,neg);
    $pwr(V(pos,neg)*I(pos,neg));
  end
endmodule
```

## Indicating Non-linearities to the Simulator

The `$limit()` function provides a method to indicate to the simulator, those nonlinearities, that are present in semiconductor device compact models other than the exponential. A user-defined function is used to limit the change of its output from iteration to iteration.

Currently `$limit()` function is supported with only `analog_function_identifier`.

You can use the `$limit()` function as below:

```
limit_call ::=
$limit ( access_function_reference , analog_function_identifier , arg_list )
```

where

`access_function_reference` is the reference that is being limited.

`analog_function_identifier` is the user-defined analog function that to be used to compute the return value.

`arg_list` is a list of arguments for user-defined function.

The `$limit()` function keep an internal state to save the previous value. A limit algorithm is implemented in user-defined function to compare the current value with the previous value, then return a limited value of access function reference. `$limit()` function then gives a flag to the simulator to indicate whether current value is limited or not. The mechanism is as same as the `limexp`, besides that `$limit()` function can use user-defined limit algorithms.

## Example

```
module diode(a,c);
  inout a, c;
  electrical a, c;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
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
```

In this release, although we support the following two syntaxes, they will directly return the expression `access_function_reference`. Other arguments will be ignored.

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
limit_call_function ::=  
    $limit ( access_function_reference )  
    | $limit ( access_function_reference, string, arg_list)
```

*access\_function\_reference* is the reference that is being limited.

*string* is a built-in simulator function that is recommended to be used to compute the return value. In this release, the syntax of *string* is not checked.

*arg\_list* is a list of arguments for the built-in function. In this release, the syntax of *arg\_list* is not checked.

Therefore, the code

```
vdio = $limit(V(a,c), spicepnjlim, $vt, vcrit);
```

is equivalent to the code

```
vdio = V(a,c);
```

## Working with Files

Verilog-A provides several functions for working with files. `$fopen` prepares a file for writing. `$fstrobe` and `$fdisplay` write to a file. `$fclose` closes an open file.

### Opening a File

Use the `$fopen` function to open a specified file.

```
fopen_function ::=  
    multi_channel_descriptor = $fopen ( "file_name" [ "io_mode" ] ) ;
```

*multi\_channel\_descriptor* is a 32-bit unsigned integer that is uniquely associated with *file\_name*. The `$fopen` function returns a *multi\_channel\_descriptor* value of zero if the file cannot be opened.

Think of *multi\_channel\_descriptor* as a set of 32 flags, where each flag represents a single output channel. The least significant bit always refers to the standard output. The first time it is called, `$fopen` opens channel 1 and returns a descriptor value of 2 (binary 10). The second time it is called, `$fopen` opens channel 2 and returns a descriptor value of 4 (binary 100). Subsequent calls cause `$fopen` to open channels 3, 4, 5, and so on, and to return values of 8, 16, 32, and so on, up to a maximum of 32 open channels.

*io\_mode* is one of three possible values: `w`, `a`, or `r`. The `w` or write mode deletes the contents of any existing files before writing to them. The `a` or append mode appends the next output to the existing contents of the specified file. In both cases, if the specified file does not exist,

## Cadence Verilog-A Language Reference

### Simulator Functions

---

`$fopen` creates that file. The `r` mode opens a file for reading. An error is reported if the file does not exist.

The `$fopen` function reuses channels associated with any files that are closed.

`file_name` is a string that can include the special commands described in “[Special \\$fopen Formatting Commands](#)” on page 192. If `file_name` contains a path indicating that the file is to be opened in a different directory, the directory must already exist when the `$fopen` function runs. `file_name` (together with the surrounding quotation marks) can also be replaced by a string parameter.

For example, to open a file named `myfile`, you can use the code

```
integer myChanDesc ;
myChanDesc = $fopen ( "myfile" ) ;
```

### Special \$fopen Formatting Commands

The following special output formatting commands are available for use with the `$fopen` function.

---

Command	Output	Example
%C	Design filename	<code>input.scs</code>
%D	Date (yy-mm-dd)	<code>94-02-28</code>
%H	Host name	<code>hal</code>
%S	Simulator type	<code>spectre</code>
%P	Unix process ID #	<code>3641</code>
%T	Time (24hh:mm:ss)	<code>15:19:25</code>
%I	Instance name	<code>opamp3</code>
%A	Analysis name	<code>dc0p, timeDomain, acSup</code>

---



## Cadence Verilog-A Language Reference

### Simulator Functions

---

The special output formatting commands can be followed by one or more modifiers, which extract information from UNIX filenames. (To avoid opening a file that is already open, the %C command must be followed by a modifier.) The modifiers are:

---

Modifier	Extracted information
:r	Root (base name) of the path for the file
:e	Extension of the path for the file
:h	Head of the path for any portion of the file before the last /
:t	Tail of the path for any portion of the file after the last /
::	The (:) character itself

---

Any other character after a colon (:) signals the end of modifications. That character is copied with the previous colon.

The modifiers are typically used with the %C command although they can be used with any of the commands. However, when the output of a formatting command does not contain a / and ".", the modifiers :t and :r return the whole name and the :e and :h modifiers return ".". As a result, be aware that using modifiers with formatting commands other than %C might not produce the results you expect. For example, using the command

```
$fopen("%I:h.freq_dat") ;
```

opens a file named `..freq_dat`.

You can use a concatenated sequence of modifiers. For example, if your design file name is `res.ckt`, and you use the statement

```
$fopen("%C:r.freq_dat") ;
```

then

- %C is the design filename (`res.ckt`)
- :r is the root of the design filename (`res`)
- .freq\_dat is the new filename extension

As a result, the name of the opened file is `res.freq_dat`.

The following table shows the various filenames generated from a design filename (%C) of

```
/users/maxwell/circuits/opamp.ckt
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

by using different formatting commands and modifiers.

---

Command and Modifiers	Resulting Opened File
<code>\$fopen("%C");</code>	None, because the design file cannot be overwritten.
<code>\$fopen("%C:r");</code>	<code>/users/maxwell/circuits/opamp</code>
<code>\$fopen("%C:e");</code>	<code>ckt</code>
<code>\$fopen("%C:h");</code>	<code>/users/maxwell/circuits</code>
<code>\$fopen("%C:t");</code>	<code>opamp.ckt</code>
<code>\$fopen("%C::");</code>	<code>/users/maxwell/circuits/opamp.ckt:</code>
<code>\$fopen("%C:h:h");</code>	<code>/users/maxwell</code>
<code>\$fopen("%C:t:r");</code>	<code>opamp</code>
<code>\$fopen("%C:r:t");</code>	<code>opamp</code>
<code>\$fopen("/tmp/%C:t:r.raw");</code>	<code>/tmp/opamp.raw</code>
<code>\$fopen("%C:e%C:r:t");</code>	<code>ckt.opamp</code>
<code>\$fopen("%C:r.%I.dat");</code>	<code>/users/maxwell/circuits/ opamp.opamp3.dat</code>

---

## Reading from a File

Use the `$fscanf` function to read information from a file.

```
fscanf_function ::=  
    $fscanf (multi_channel_descriptor , "format" { , storage_arg } )
```

The *multi\_channel\_descriptor* that you specify must have a value that is associated with one or more currently open files. The format describes the matching operation done between the `$fscanf` storage arguments and the input from the data file. The `$fscanf` function sequentially attempts to match each formatting command in this string to the input coming from the file. After the formatting command is matched to the characters from the input stream, the next formatting command is applied to the next input coming from the file. If a formatting command is not a skipping command, the data read from the file to match a formatting command is stored in the formatting command's corresponding *storage\_arg*. The first *storage\_arg* corresponds to the first nonskipping formatting command; the second *storage\_arg* corresponds to the second nonskipping formatting command. This matching process is repeated between all formatting commands and input data. The

formatting commands that you can use are the same as those used for `$strobe`. See [“\\$strobe”](#) on page 184 for guidance.

For example, the following statement reads data from the file designated by `fptr1` and places the information in variables called `dbl` and `int`.

```
$fscanf(fptr1, "Double = %e and Integer = %d", dbl, int);
```

## Writing to a File

Verilog-A provides three input/output functions for writing to a file: `$fstrobe`, `$fdisplay`, and `$fwrite`. The `$fstrobe` and `$fdisplay` functions use the same arguments and are completely interchangeable. The `$fwrite` function is similar but does not insert automatic carriage returns in the output.

### **\$fstrobe**

Use the `$fstrobe` function to write information to a file.

```
fstrobe_function ::=
    $fstrobe (multi_channel_descriptor {,list_of_arguments })
list_of_arguments ::=
    argument
    | list_of_arguments , argument
```

The *multi\_channel\_descriptor* that you specify must have a value that is associated with one or more currently open files. The arguments that you can use in *list\_of\_arguments* are the same as those used for `$strobe`. See [“\\$strobe”](#) on page 184 for guidance.

For example, the following code fragment illustrates how you might write simultaneously to two open files.

```
integer mcd1 ;
integer mcd2 ;
integer mcd ;
@(initial_step) begin
    mcd1 = $fopen("file1.dat") ;
    mcd2 = $fopen("file2.dat") ;
end
.
.
.
mcd = mcd1 | mcd2 ; // Bitwise OR combines two channels
$fstrobe(mcd, "This is written to both files") ;
```

## **\$display**

Use the `$display` function to write information to a file.

```
fdisplay_function ::=  
    $display (multi_channel_descriptor {,list_of_arguments })  
list_of_arguments ::=  
    argument  
    | list_of_arguments , argument
```

The *multi\_channel\_descriptor* that you specify must have a value that is associated with a currently open file. The arguments that you can use in `list_of_arguments` are the same as those used for `$strobe`. See “[\\$strobe](#)” on page 184 for guidance.

## **\$fwrite**

Use the `$fwrite` function to write information to a file.

```
fwrite_function ::=  
    $fwrite (multi_channel_descriptor {,list_of_arguments })  
list_of_arguments ::=  
    argument  
    | list_of_arguments , argument
```

The *multi\_channel\_descriptor* that you specify must have a value that is associated with a currently open file. The arguments that you can use in `list_of_arguments` are the same as those used for `$strobe`. See “[\\$strobe](#)” on page 184 for guidance.

The `$fwrite` function does not insert automatic carriage returns in the output.

## **\$fflush**

Use the `$fflush` function to flush any buffered output to a file.

```
file_flush_function ::=  
    $fflush ( multi_channel_descriptor ) ;
```

The `multi_channel_descriptor` that you specify must have a value that is associated with the currently open file.

### **Example:**

```
$fflush ( mcd );  
$fflush ( fd );
```

Writes any buffered output to the file(s) specified by `mcd` or to the file specified by `fd`.

The `$fflush()` function with empty argument is not supported.

## Closing a File

Use the `$fclose` function to close a specified file.

```
file_close_function ::=  
    $fclose ( multi_channel_descriptor ) ;
```

The *multi\_channel\_descriptor* that you specify must have a value that is associated with the currently open file that you want to close.

## \$feof

Use the `$feof` function to return a nonzero value when End of File (EOF) has previously been detected reading the input file `fd`:

```
integer code;  
code = $feof ( fd );
```

If the EOF is not detected previously, a value of zero is returned.

## Writing to a Variable

### \$swrite

Use the `$swrite` function to store the output in a variable. The `$swrite` function is similar to the `$fwrite` function (see [\\$fwrite](#) on page 196). However, the first argument to the `$swrite` function is a string variable in which resulting string is stored.

```
string_output_task ::=  
    $swrite ( string_variable , list_of_arguments );
```

### Example

```
strSrc = "string1";  
$swrite( strDes, "swrite: %s", strSrc );
```

In the above example, the output of `strDes` would be “swrite: string1”.

### \$sformat

Use the `$sformat` function to store the output in a variable. The `$sformat` function is similar to `$swrite` except that the `$sformat` function always interprets only its second

argument as a format string. In addition, the argument can be a static string, such as “data is %d” or a string variable whose content is interpreted as the format string.

**Note:** If `format_string` is a string variable, you might not be able to determine its value at compile time.”

The other arguments to the `$sformat` function are processed using the format specifiers in `format_string`. If you do not specify sufficient arguments or specify too many arguments, the application generates a warning but continues execution.

```
variable-format_string_output_task ::=  
    $sformat ( string_variable , format_string , list_of_arguments );
```

### Example

```
strSrc = "string1";  
formatStr = "sformat: %s"  
$sformat( strDes, formatStr, strSrc );
```

In the above example, the output of `strDes` would be “sformat: string1”.

## Simulator Control Functions

Verilog-A provides the following simulator control functions:

- [\\$finish](#) on page 198
- [\\$stop](#) on page 199
- [\\$fatal](#) on page 200
- [\\$error](#) on page 201
- [\\$warning](#) on page 201
- [\\$info](#) on page 201

**Note:** For simulation tools, the `$fatal`, `$error`, `$warning`, and `$info` functions also report the simulation run time at which the function is called. You can also use these functions to include additional information using the same format as the `$display` function.

### \$finish

Use the `$finish` function to make the simulator exit and return control to the operating system.

```
finish_function ::=  
    $finish [( msg_level )] ;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
msg_level ::=
    0 | 1 | 2
```

The `msg_level` value determines which diagnostic messages print before control returns to the operating system. The default `msg_level` value is 1.

---

<b>msg_level</b>	<b>Messages printed</b>
0	None
1	Simulation time and location
2	Simulation time, location, and statistics about the memory and CPU time used in the simulation

---

**Note:** In this release, the `$finish` function always behaves as though the `msg_level` value is 0, regardless of the value you actually use.

For example, to make the simulator exit, you might code

```
$finish ;
```

### **\$finish\_current\_analysis**

Use the `$finish_current_analysis` function to stop the current transient analysis without exiting the simulator. The `$finish_current_analysis` enables you to stop the ongoing transient analysis and continue to run the remaining analysis. The syntax of `$finish_current_analysis` is the following:

```
finish_function ::=
$finish_current_analysis [( msg_level )] ;
msg_level ::=
    0 | 1 | 2
```

where `msg_level` value works exactly the same way as it does in the `$finish` function.

**Note:** The `$finish_current_analysis` can be used to stop only the transient analysis and not any other analysis type.

### **\$stop**

Use the `$stop` function to make the simulator enter interactive mode and display a Tcl prompt.

```
stop_function ::=
    $stop [( msg_level )] ;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
msg_level ::=
    0 | 1 | 2
```

The `msg_level` value determines which diagnostic messages print before the simulator starts the interactive mode. The default `msg_level` value is 1.

---

<b>msg_level</b>	<b>Messages printed</b>
0	None
1	Simulation time and location
2	Simulation time, location, and statistics about the memory and CPU time used in the simulation

---

For example, to make the simulator go interactive, you might code

```
$stop ;
```

### **\$fatal**

Use the `$fatal` function to generate a runtime fatal error message and make the simulator exit with an error code. The first argument to the `$fatal` function is the same as the `$finish` function and identifies which diagnostic message should the simulator print before exiting.

```
fatal_function ::=
    $fatal [ ( msg_level [ , message_argument { , message_argument } ] ) ] ;
```

```
msg_level ::=
    0 | 1 | 2
```

---

<b>msg_level</b>	<b>Messages printed</b>
0	None
1	Simulation time and location
2	Simulation time, location, and statistics about the memory and CPU time used in the simulation

---

**Note:** Currently, the `$fatal` function, like the `$finish` function, behaves as though `msg_level` is 0 regardless of the value that you specify for `msg_level`.



## **\$error**

Use the `$error` function to generate a runtime error message.

```
error_function ::=  
    $error [ ( [ message_argument { , message_argument } ] ) ] ;
```

## **\$warning**

Use the `$warning` function to generate a runtime warning message, which can be suppressed by the simulator.

```
warning_function ::=  
    $warning [ ( [ message_argument { , message_argument } ] ) ] ;
```

## **\$info**

Use the `$info` function to generate an informational message that does not include any severity.

```
info_function ::=  
    $info [ ( [ message_argument { , message_argument } ] ) ] ;
```

**Note:** The `$fatal`, `$error`, `$warning`, and `$info` functions also report the simulation run time at which the severity system task is called. In addition, you can use these functions to include additional information using the same format as the `$display` function.

# **Obtaining the Trial Number for Monte Carlo Analysis**

## **\$cds\_get\_mc\_trial\_number()**

Use the `$cds_get_mc_trial_number` function to obtain the trial number for the Monte Carlo analysis. For example,

```
module test(p);  
    inout p;  
    electrical p;  
    integer trial_num;  
  
    analog begin
```

```
    trial_num = $cdfs_get_mc_trial_number();  
    $strobe("Monte Carlo trial number = %d", trial_num );  
end  
endmodule
```

## User-Defined Functions

Verilog-A supports user-defined functions. By defining and using your own functions, you can simplify your code and enhance readability and reuse. See the following topics for more information:

- [Declaring an Analog User-Defined Function](#) on page 203
- [Calling a User-Defined Analog Function](#) on page 207

## Declaring an Analog User-Defined Function

To define an analog function, use this syntax:

```
analog_function_declaration ::=
    analog function [ function_type ] function_identifier ;
    function_item_declaration {function_item_declaration}
    function_statement
    endfunction

function_type ::=
    integer
    | real

function_item_declaration ::=
    input_declaration;
    | output_declaration;
    | inout_declaration;
    | block_item_declaration;

block_item_declaration ::=
    integer_declaration
    | real_declaration
```

An analog user-defined function declaration should begin with the keyword `analog function`, optionally followed by the type of the return value from the function, the name of the function and a semicolon, and should end with the keyword `endfunction`.

`function_type` is optional and specifies the return value of the function. `type` can be a real or an integer. If it is unspecified, the default is real.

An analog user-defined function:

- should not include analog operators.
- should not define module behavior.
- should not use access functions.
- should not use contribution statements or event control statements.
- should not use analog filter functions.
- should have at least one input declared; the block item declaration should declare the type of the inputs ( `input`, `output`, `inout`) as well as local variables used in the function.
- should not use named blocks.
- should only reference locally-defined variables, variables passed as arguments and module-level parameters.
- can use `$vt`, `$vt(temp)`, `$temperature`, and `$abstime`
- can use analysis

## Cadence Verilog-A Language Reference

### Simulator Functions

---

- can use \$strobe, \$display, \$write, \$fopen, \$fstrobe, \$fdisplay, \$fwrite and \$fclose
- can use all mathematical functions

#### **Example 1**

The following example defines an analog user-defined function called `chopper`, which takes two variables, `sw` and `in`, assign a value to the implicitly defined internal variable with the same name as the function and returns a real result.

```
analog function real chopper ;
    input sw, in ; // The function has two declared inputs.
    real sw, in ;
    begin
        //The next line assigns a value to the implicit variable, chopper.
        chopper = ((sw > 0) ? in : -in) ;
    end
endfunction
```

#### **Example 2**

The following 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;
        perim = 2 * ( l + w );
    end
endfunction
```

#### **Example 3**

The following example defines an analog user-defined function `arrayadd` that adds the contents of second array to the first array.

```
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
```

#### **Example 4**

The following example defines a user-defined analog function `mult` that accesses the module-level parameters.

```
module A(a);
parameter real p1 = 1;
analog function real mult;
    real a;
input a;
begin
    mult = a * 2 * p1 ; // access module-level parameter p1
end
endfunction
endmodule
```

### **Returning a Value From an Analog User-Defined Function**

There are three methods to return a value from an analog user-defined function:

- Using the implicit analog user-defined function identifier variable
- Using an `output` argument
- Using an `inout` argument.

#### **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 `function_identifier`. This variable inherits the same type as the type specified in the analog user-defined function declaration. The internal variable is initialized to zero (0) and can be used within the body of the analog user-defined function. The last value assigned to this variable is the return value

of the analog user-defined function. If the internal variable is not assigned during the execution of the analog user-defined function, then the analog user-defined function returns the initialized value zero (0).

The following line ( in Example 1 ) illustrates this concept:

```
chopper = ((sw > 0) ? in : -in) ;
```

### Output Argument

The `output` argument allows you to return more than one values. The argument passed to the `output` argument must be an analog variable reference. If the `output` argument is defined as an array, then the argument passed in the function must be an analog variable. All `output` arguments of an analog user-defined function are initialized to zero (0), which means that the argument passed to the function is reset to zero (0). When the analog user-defined function is run, these variables can be read and assigned in the flow. After the function is run, the last value assigned to the `output` argument is then assigned to the corresponding analog variable reference that was passed to the function.

The following lines ( in Example 2 ) illustrate this concept:

```
area = l * w;  
perim = 2 * ( l + w );
```

### Inout Arguments

The `inout` arguments allow you to pass 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 to the function must be an analog variable. The `inout` arguments of an analog user-defined function do not get initialized to zero (0). When the analog user-defined function is run, these variables can be read and assigned in the flow. After the function is run, the last value assigned to the `inout` argument is then assigned to the corresponding analog variable reference that was passed to 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 not changed.

The following lines ( in Example 3 ) illustrate the use of the `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.

## Calling a User-Defined Analog Function

To call a user-defined analog function, use the following syntax.

```
analog_function_call ::=  
    function_identifier ( expression { , expression } )
```

*function\_identifier* must be the name of a defined function. The order of evaluation of the arguments of 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 function must not call itself, either directly or indirectly, because recursive functions are illegal. Analog function calls are allowed only inside analog blocks.

The module `phase_detector` illustrates how the `chopper` function can be called.

```
module phase_detector(lo, rf, if0) ;  
module phase_detector(lo, rf, if0) ;  
inout lo, rf, if0 ;  
electrical lo, rf, if0 ;  
parameter real gain = 1 ;  
    function real chopper;  
        input sw, in;  
        real sw, in;  
    begin  
        chopper = ((sw > 0) ? in : -in);  
    end  
endfunction  
  
analog begin  
    V(if0) <+ gain * chopper(V(lo),V(rf)); //Call user-defined function  
end  
endmodule
```

The following example uses the `geomcalc` function defined in Example 2.

```
geomcalc(l-dl, w-dw, ar, per);
```

**Note:** The first two arguments are expressions, and match with the inputs `l` and `w` for the function. The second two arguments must be real variables because they match with the function outputs.

The following example incorrectly uses the `geomcalc` function defined in Example 2.

```
geomcalc(l-dl, w-dw, ar, V(a));
```

Here, the last two arguments to the user-defined function `geomcalc` are declared as `output` arguments, however, the fourth argument is passed the potential probe `V(a)`. Only analog variable references can be passed to the `output` and `inout` arguments of the analog user-defined function. Therefore the above example will generate an error.

The following example uses the `arrayadd` example defined in Example 3, to add values of one array to another.

```
x[0] = 5; x[1] = 10;  
y[0] = 3; y[1] = 6;  
arrayadd(x, y);
```

Here, the first and second arguments are both expecting vectors. Two vector variables are passed to them. Since the first argument is an `inout` argument, the result of calling the `arrayadd` function will update the vector variable `x` with values `x[0] = 8` and `x[1] = 16`.

## Calling functions implemented in C

Verilog-A supports calling functions implemented in C and imported from a shared library.

### Import declaration

Each C function which is called in verilog-A must be declared. Such declaration is referred to as import declaration. The syntax for import declaration is shown below:

```
c_import_declaration ::=  
    import "CDS_VA_DPI" function type function_identifier(function_param_list {,  
        derivative_declaration} );  
type ::= =  
    integer|real  
function_param_list ::= =  
    param_declaration {, param_declaration}  
param_declaration ::= =  
    integer variable_identifier|real variable_identifier  
derivative_declaration ::= =  
    ddx(*)
```

An import declaration begins with the keywords **import "CDS\_VA\_DPI" function**, followed by the type of the return value from the function, then the function name, function parameters and optional followed by the derivatives declaration, and ending with a semicolon.



*Type* specifies the return value and parameters of the function. It can be a real or an integer, corresponding the double and int in C.

The *partial derivatives* of the function return value with respect of all function parameters that may be provided by the C function. In such a case, use derivative declaration `ddx(*)` followed the function parameter list in the import declaration. Import declaration shall be within a verilog-A module.

The following are examples of import declaration:

```
import "CDS_VA_DPI" function real c_noparam();
import "CDS_VA_DPI" function real c_add( real a, real b);
import "CDS_VA_DPI" function real c_add_deriv( real a, real b, ddx(*));
```

## Loading C function from a Dynamic Link Library

The C functions called by verilog-A are loaded from a shared library.

### Create functions in 'C'

The C function interface is consistent with the import declaration in verilog-A. If partial derivatives of return value is provided by the C function (use derivatives declaration in import declaration in verilog-A), there are additional parameters in C function with type (double \*) to return these derivatives.

### Examples

#### c function which does not provide derivatives

```
/*
 * c_add_deriv.c, corresponding the followed import declaration.
 * import "CDS_VA_DPI" function real c_add_deriv( real a, real b)
 */

double c_add_deriv( double a , double b)
{
    double ret = 5*a + 8*b;
    return ret;
}
```

#### c function which provides derivatives

```
/*
 * c_add_deriv.c, corresponding the followed import declaration.
 * import "CDS_VA_DPI" function real c_add_deriv( real a, real b, ddx(*))
 */
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
double c_add_deriv( double a , double b, double *d_a, double *d_b)
{
    double ret = 5*a + 8*b;
    *d_a = 5;      /* (*d_a) = d(ret)/d(a) */
    *d_b = 8;      /* (*d_b) = d(ret)/d(b) */
    return ret;
}
```

### Compiling the C functions into Dynamic Shared Library

A script can be used to automatically generate the GNUmakefile and compile the C function into a shared library. The script is installed as ``${MMSIM_ROOT}/bin/mmsim_genplugin`.

Given a C file `c_add_deriv.c`, and the library name `libfunc_sh.so`, the following invocation of `mmsim_genplugin` generates the GNUmakefile and compiles the c file into the shared library on all supported platforms.

```
mmsim_genplugin -n func -b c_add_deriv.c
```

use `mmsim_genplugin -h` to see more usage of `mmsim_genplugin`.

### Loading Dynamic Link Library

User the following command in the circuit file to indicate which library will be loaded into verilog-A:

```
ahdl_include "library_name" -cvs_va_dpi
```

### Example

```
// c_add_deriv.ckt
global gnd
simulator lang=spectre
ahdl_include "%C:r.va"
vin in gnd vsource type=sine ampl=1 freq=100kHz
s1 in out c_add_mod
timeDomain1 tran stop=.1m
ahdl_include "../libcfunc_sh.so" -cvs_va_dpi
// c_add_deriv.va
`include "discipline.vams"
`include "constants.vams"
module c_add_mod(in, out);
input in, out;
electrical in, out;
```

## Cadence Verilog-A Language Reference

### Simulator Functions

---

```
import "CDS_VA_DPI" function real c_add_deriv( real a, real b, ddx(*))
    analog begin
        V(out) <+ c_add(V(in), V(in));
    end
endmodule
```

# Cadence Verilog-A Language Reference

## Simulator Functions

---

---

## Instantiating Modules and Primitives

---

Chapter 2, “[Creating Modules](#),” discusses the basic structure of Cadence® Verilog®-A language modules. This chapter discusses how to instantiate Verilog-A modules within other modules. You must not nest module declarations in one another; instead, you embed instances of modules in other modules. By embedding instances, you build a hierarchy extending from the instances of primitive modules up through the top-level modules.

- For information about instantiating modules in Spectre® circuit simulator netlists, see [Appendix 19, “Getting Ready to Simulate.”](#)
- For information about instantiating a Verilog-A module in a schematic or a schematic in a Verilog-A module, see [“Multilevel Hierarchical Designs”](#) on page 285.

See the following topics for more information:

- [Instantiating Verilog-A Modules](#) on page 214
- [Connecting the Ports of Module Instances](#) on page 219
- [Overriding Parameter Values in Instances](#) on page 220
- [Instantiating Analog Primitives](#) on page 225
- [Using Inherited Ports](#) on page 226
- [Using an Inherited m Factor \(Multiplicity Factor\)](#) on page 228
- [Setting an m Factor Directly on a Verilog-A Module](#) on page 230
- [Using the \\$mfactor System Function](#) on page 232

## Instantiating Verilog-A Modules

Use the following syntax to instantiate modules in other modules.

```
module_instantiation ::=
    module_or_paramset_id [ parameter_value_assignment ] instance_list
instance_list ::=
    module_instance { , module_instance } ;
module_instance ::=
    name_of_instance ( [ list_of_module_connections ] )
name_of_instance ::=
    module_instance_identifier
list_of_module_connections ::=
    ordered_port_connection { , ordered_port_connection }
    | named_port_connection { , named_port_connection }
ordered_port_connection ::=
    [ net_expression ]
    | { net_expression { , net_expression } }
named_port_connection ::=
    . net_identifier ( net_expression )
    | . net_identifier ( { net_expression { , net_expression } } )
net_expression ::=
    net_identifier
    | net_identifier [ constant_expression ]
    | net_identifier [ constant_range ]
constant_range ::=
    constant_expression : constant_expression
```

The `instance_list` expression is discussed in the following sections. The `parameter_value_assignment` expression is discussed in [“Overriding Parameter Values in Instances”](#) on page 220.

### Creating and Naming Instances

This section illustrates how to instantiate modules. Consider the following module, which describes a gain block that doubles the input voltage.

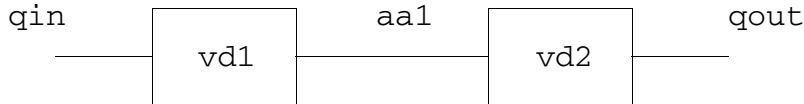
```
module vdoubler (in, out) ;
input in ;
output out ;
electrical in, out ;
analog
    V(out) <+ 2.0 * V(in) ;
endmodule
```

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

Two of these gain blocks are connected, with the output of the first becoming the input of the second. The schematic looks like this.



This higher-level component is described by module `vquad`, which creates two instances, named `vd1` and `vd2`, of module `vdoubler`. Module `vquad` also defines external ports corresponding to those shown in the schematic.

```
module vquad (qin, qout) ;
input qin ;
output qout ;
electrical qin, qout ;
wire aa1 ;
vdoubler vd1 (qin, aa1) ;
vdoubler vd2 (aa1, qout) ;
endmodule
```

## Mapping Instance Ports to Module Ports

When you instantiate a module, you must specify how the actual ports listed in the instance correspond to the formal ports listed in the defining module. Module `vquad`, in the previous example, uses an ordered list, where instance `vd1`'s first actual port name `qin` maps to `vdoubler`'s first formal port name `in`. Instance `vd1`'s second actual port name `aa1` maps to `vdoubler`'s second formal port name, and so on.

## Mapping Ports By Order

To use ordered lists to map actual ports listed in the instance to the formal ports listed in the defining module, ensure that the instance ports are in the same order as the defining module ports. For example, consider the following module `child` and the module `instantiator` that instantiates it.

```
module child (ina, inb, out) ;
input [0:3] ina ;
input inb ;
output out ;
electrical [0:3] ina ;
electrical inb ;
electrical out ;
endmodule

module instantiator (conin, conout) ;
input [0:6] conin ;
output conout ;
electrical [0:6] conin ;
electrical conout ;
```

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

```
child child1 (conin [1:4], conin [6], conout) ;
end module
```

You can tell from the order of port names in these modules that port `ina[0]` in module `child` maps to port `conin[1]` in instance `child1`. Similarly, port `inb` in `child` maps to port `conin[6]` in instance `child1`. Port `out` in `child` maps to port `conout` in instance `child1`.

### Mapping Ports By Name

Mapping module ports consists of explicitly linking the two names for each side of the connection; the port declaration name from the module declaration to the expression, that is, the name used in the module declaration, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections. The port name is the name specified in the module declaration. The port name cannot be a bit-select, a part-select, or a concatenation of ports. If the module port declaration is implicit, the port expression should be a simple identifier or escaped identifier, which is used as the port name. If the module port declaration is explicit, the explicit name is used as the name of port. The port expression can be any valid expression.

**Note:** The two types of module port connections shall not be mixed; connections to the ports of a particular module instance shall be all by order or all by name.

Consider the following examples.

#### Example 1

In the following example, the instantiating module connects its signals `topA` and `topB` to the ports `In1` and `Out` defined by the module `ALPHA`.

```
ALPHA instance1 (.Out(topB), .In1(topA));
```

#### Example 2

The following example defines the modules `modB` and `topmod`. The module `topmod` instantiates `modB` using the ports connected by name. Because these connections are made by name, the order in which they appear is irrelevant.

```
module modB(wa, wb, c, d);
  inout wa, wb;
  input c, d;
  .....
endmodule

module topmod;
  inout [2:0] v;
  inout w;
  modB b1 (.wb(v[1]), .wa(v[0]), .d(v[2]), .c(w));
endmodule
```



endmodule

### Declaring Multiple Instances of a Module as an Array

You can specify multiple instances of a module as an instance array. To specify an array of instances, the instance name should be followed by a range specification. The range is specified using two constant expressions, separated by a colon and enclosed within square brackets. If both constant expressions are equal, only one instance is generated.

The range specification represents an array of module instances, which should have a continuous range. To declare an array of instances, one instance identifier should be associated with one range only.

The connection list of an instance array should be enclosed in parentheses and the terminals should be separated by a comma. The following rules apply to the terminal connections for an array of instances:

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

**Note:** The range specification for instances array and ports should be constant expressions. Parameters cannot be used to define constant expressions.

The following are some examples of a module instance array:

#### Example1

```
module moduleA (a, b);
  inout [0:1]a;
  inout b;
  .....
endmodule
module moduleB (c, d);
  inout [0:5]c;
  inout d;
moduleA INST[0:2] (c, d); // instance array
endmodule
```

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

In the above example, an instance array `INST[0:2]` has been used in `module B` to instantiate `module A`. The module instance can be referenced by `INST[0]`, `INST[1]`, and `INST[2]` respectively.

#### Example2

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

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

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

endmodule

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

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

endmodule
```

#### Example3

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

```
module res_mul (vp1, vp2, enh, enl, vn);
inout [0:3] vp1, vp2;
inout [0:11] vn;
input enh, enl;

// connect ports by order
res INST_RES[0:3]({vp1, vp2}, {enh, enh, enl, enl}, vn);
```

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

```
endmodule
module res_mul_equiv (vp1, vp2, enh, en1, vn);
  inout [0:3] vp1, vp2;
  inout [0:11] vn;
  input enh, en1;

  res INST_RES0 (vp1[0:1], enh, vn[0:2]);
  res INST_RES1 (vp1[2:3], enh, vn[3:5]);
  res INST_RES2 (vp2[0:1], en1, vn[6:8]);
  res INST_RES3 (vp2[2:3], en1, vn[9:11]);

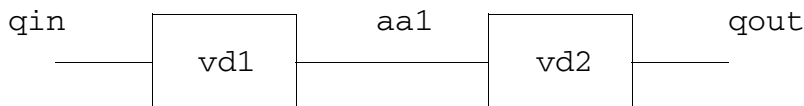
endmodule
```

**Note:** If the instantiated module and the module which includes the declared instance array are defined in different Verilog-A files, you should first specify the Verilog-A file which is used to define the instantiated module in the netlist. For example, if Verilog-A file `moduleA.va` includes the definition of module A and Verilog-A file `moduleB.va` includes the definition of module B which instantiates module A, then you should first specify `moduleA.va` in the netlist file.

## Connecting the Ports of Module Instances

Developing modules that describe components is an important step to achieve the overall goal of simulating a system. However, an equally important step is to combine those components so that they represent the system as a whole. This section discusses how to connect module instances, using their ports, to describe the structure and behavior of the system you are modeling.

Consider again the modules `vdoubler` and `vquad`, which describe this schematic.



```
module vdoubler (in, out) ;
  input in ;
  output out ;
  electrical in, out ;
  analog
    V(out) <+ 2.0 * V(in) ;
endmodule

module vquad (qin, qout) ;
  input qin ;
  output qout ;
```

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

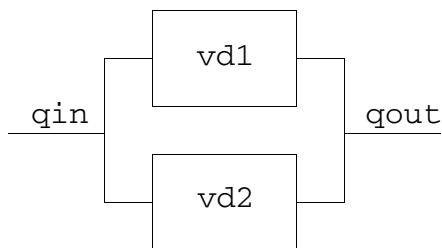
```
electrical qin, qout ;  
wire aa1 ;  
vdoubler vd1 (qin, aa1) ;  
vdoubler vd2 (aa1, qout) ;  
endmodule
```

This time, note how the module instance statements in `vquad` use port names to establish a connection between output port `aa1` of instance `vd1` and input port `aa1` of instance `vd2`.

Module instance statements like

```
vdoubler vd1 (qin, qout) ;  
vdoubler vd2 (qin, qout) ;
```

establish different connections. These statements describe a system where the gain blocks are connected in parallel, with this schematic.



## Port Connection Rules

You can connect the ports described in the `vdoubler` instances because the ports are defined with compatible disciplines and are the same size. To generalize,

- You must ensure that all ports connected to a net are compatible with each other. Ports of any analog discipline are compatible with a reference node (ground). For a discussion of compatibility, see [“Compatibility of Disciplines”](#) on page 72.
- You must ensure that the sizes of connected ports and nets match. In other words, you can connect a scalar port to a scalar net, and a vector port to a vector net or concatenated net expression of the same width.

## Overriding Parameter Values in Instances

As noted earlier, the syntax for the module instance statement is

```
module_or_paramset_id [ parameter_value_assignment ] instance_list
```

The following sections discuss the `parameter_value_assignment` expression, which is further defined as

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

```
parameter_value_assignment ::=  
    | #( named_param_override_list )  
named_param_override_list ::=  
    named_param_override { , named_param_override }  
named_param_override ::=  
    . parameter_identifier ( expression )
```

By default, instances of modules inherit any parameters specified in their defining module. If you want to change any of the default parameter values, you do so on the module instance statement itself. You can also use paramsets, as described in [“Overriding Parameter Values by Using Paramsets”](#) on page 223.

## Overriding Parameter Values from the Module Instance Statement

Using the module instance statement, you can assign values to parameters by explicitly referring to parameter names. The new values must be constant expressions. The format for overriding a parameter value on an instance statement is as follows:

```
moduleName # (.parameterName(constantExpression)) instanceName (ports) ;
```

For example:

```
vdoubler # (.parm3(4.0)) vd1 (qin, aa1) ;
```

You only need to name those parameters whose values you want to override.

Consider the following module `vdoubler` definition, which has three parameters: `parm1`, `parm2`, and `parm3`.

```
module vdoubler (in, out) ;
input in ;
output out ;
electrical in, out ;
parameter parm1 = 0.2,
           parm2 = 0.1,
           parm3 = 5.0 ;
analog
    V(out) <+ (parm1 + parm2 + parm3) * V(in) ;
endmodule
```

The `vdoubler` module instance statements in the following module illustrate how you can override any of these parameters, by name. For example, instance `vd1` of the `vdoubler` module overrides the value of parameter `parm3`, by name, to 4.0. For this instance, the other two parameters retain their default values, 0.2 for `parm1` and 0.1 for `parm2`.

```
module vquad (qin, qout) ;
input qin ;
output qout ;
vdoubler # (.parm3(4.0)) vd1 (qin, aa1) ; // Overriding by name
vdoubler # (.parm1(0.3), .parm2(0.2)) vd2 (aa1, qout) ; // Overriding by name
vdoubler # (.parm1(0.3), .parm2(0.2)) vd3 (aa1, qout) ; // By name
endmodule
```

## Overriding Parameter Values by Using Paramsets

See “[Paramsets](#)” on page 64 for information about the syntax for creating paramsets. This section discusses how to use paramsets to override parameter values.

The syntax for module instantiation is

```
module_instantiation ::=  
    module_or_paramset_id [ parameter_value_assignment ] instance_list
```

According to this syntax, the paramset can be instantiated instead of a module. Because the paramset references a module, all the information contained in the module is available. For example, consider the following module and paramset definitions.

```
module baseModule (in, out);  
    inout in, out;  
    electrical in, out;  
    parameter real a = 0;  
    parameter real b = 0;  
    parameter real c = 0;  
    (* desc="output variable o1" *) real o1;  
    (* desc="output variable o2" *) real o2;  
    analog begin  
        V(out) <+ (a+b+c)*V(in);  
        baseOutput = a+b+c;  
    end  
endmodule  
  
paramset ps baseModule;  
parameter real a = 1.0 from [0:1];  
parameter real b = 1.0 from [0:1];  
.a = a; .b = b;  
endparamset  
  
paramset ps baseModule;  
parameter real b = 2.0 from (1:2];  
parameter real c = 1.0 from [0:1];  
.b = b; .c = c;  
endparamset
```

Two paramsets named `ps` are defined, and, as required, both paramset declarations reference the same module, `baseModule`.

In the following code, the paramset is instantiated in place of the referenced module. For instance `inst1`, the simulator selects the second paramset named `ps`, because that paramset declares a range of `[1:2]` for the `b` value and instance `inst1` specifies a parameter `b` value of 1.5, which is included in that range.

```
// instantiation  
ps #(.b(1.5)) inst1 (in, out);
```

The value 1.5 for parameter `b` overrides the parameter value 0 specified in `baseModule`.

The simulator uses the following rules to choose a paramset from among those with the specified name:

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

- All parameters overridden on the instance must be parameters of the paramset.
- The parameters of the paramset, with overrides and defaults, must all be within the allowed ranges specified in the paramset parameter declaration.
- The local parameters of the paramset, computed from parameters, must be within the allowed ranges specified in the paramset.

If the preceding rules are not sufficient to pick a unique paramset, the following rules are applied in order until a unique paramset is selected:

1. The paramset that has the fewest number of un-overridden parameters is selected.
2. The paramset that has the greatest number of local parameters with specified ranges is selected.

It is an error if more than one applicable paramset remains for an instance after these rules are applied.

Instances of paramsets are allowed to override only parameters that are declared in the paramset. Using a paramset instance to attempt to override a parameter of the base module that is not declared in the paramset results in a warning and the offending parameter override is ignored.



## Instantiating Analog Primitives

The remaining sections of the chapter describe how to instantiate some analog primitives in your code. For more information, see the “Preparing the Design: Using Analog Primitives and Subcircuits” chapter of the *Virtuoso AMS Designer simulator User Guide*.

As you can instantiate Verilog-A modules in other Verilog-A modules, you can instantiate Spectre and SPICE masters in Verilog-A modules. You can also instantiate models and subcircuits in Verilog-A modules. For example, the following Verilog-A module instantiates two Spectre primitives: a resistor and an isource.

```
module ri_test (pwr, gnd) ;
electrical pwr, gnd ;
parameter real ibias = 10u, ampl = 1.0 ;
electrical in, out ;
    resistor #(.r(100K)) RL (out, pwr) ;           //Instantiate resistor
    isource #(.dc(ibias)) Iin (gnd, in) ;        //Instantiate isource
endmodule
```

When you connect a net of a discrete discipline to an analog primitive, the simulator automatically inserts a connect module between the two.

However, some instances require parameter values that are not directly supported by the Verilog-A language. The following sections illustrate how to set such values in the instance statement.

## Instantiating Analog Primitives that Use Array Valued Parameters

Some analog primitives take array valued parameters. For example, you might instantiate the `svcvs` primitive like this:

```
module fm_demodulator(vin, vout, vgn) ;
input vin, vgn ;
output vout ;
electrical vin, vout, vgn ;
parameter real gain = 1 ;
    svcvs #(.gain(gain), .poles({-1M, 0, -1M, 0}))
        af_filter (vout, vgn, vin, vgn) ;
    analog begin
        ...
    end
endmodule
```

This `fm_demodulator` module sets the array parameter `poles` to a comma-separated list enclosed by a set of square brackets.

## Instantiating Modules that Use Unsupported Parameter Types

Spectre built-in primitives take parameter values that are not supported directly by the Verilog-A language. The following cases illustrate how to instantiate such modules.

To set a parameter that takes a string type value, set the value to a string constant. For example, the next fragment shows how you might set the `file` parameter of the `vsource` device.

```
vsource #(.type("pwl"), .file("mydata.dat")) V1(src,gnd);
```

To set an enumerated parameter in an instance of a Spectre built-in primitive, enclose the enumerated value in quotation marks. For example, the next fragment sets the parameter `type` to the value `pulse`.

```
vsource #(.type("pulse"),.val1(5),.period(50u)) Vclk(clk,gnd);
```

## Using Inherited Ports

The Cadence implementation of the Verilog-A language supports inherited terminals. Often, the inherited terminals arise from netlisting inherited ports in the Virtuoso Schematic Composer but you can also code inherited terminals by hand in a Verilog-A module.

The Cadence analog design environment translates the inherited terminals among the tools in the flow. For example, in the CIW, you select *File – New – Cellview* and create the following Verilog-A cellview.

```
// VerilogA for amslib, inv, veriloga
`include "constants.vams"
`include "discipline.vams"
module inv(out, gnd, vdd, in);
output out;
electrical out;

(* inh_conn_prop_name = "gnd",
   inh_conn_def_value = "cds_globals.\\gnd!" *) inout gnd;

electrical gnd;
(* inh_conn_prop_name="vdd",
   inh_conn_def_value = "cds_globals.\\vdd! " *) inout vdd;

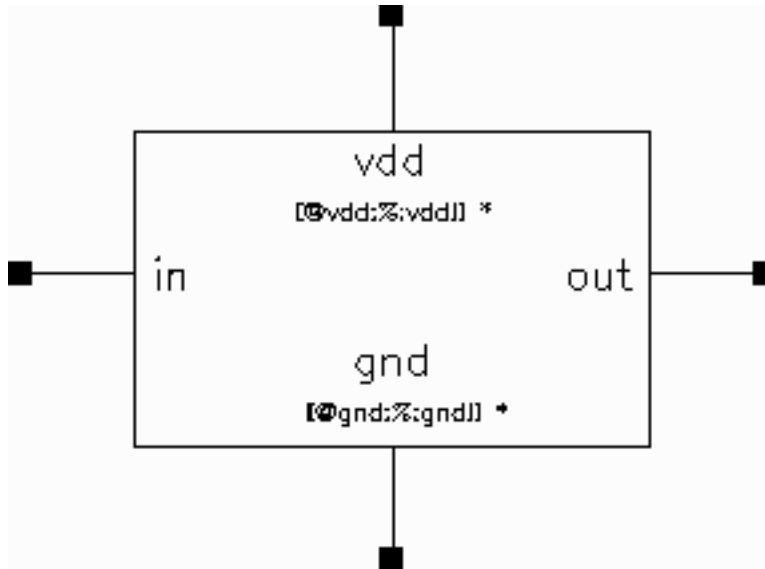
electrical vdd;
input in;
endmodule
```

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

When you save the module, you request the automatically generated symbol, which looks like this. The inherited terminal properties are automatically associated with the terminals in the symbol.



The inverse is also true. If you create a Verilog-A module from a symbol that contains inherited terminal information, the template for the new module contains the inherited terminal information.

Be aware that if you use Verilog-A without the environment, inherited terminals are not supported. Inherited nets and the netSet properties are not supported.

## Using an Inherited m Factor (Multiplicity Factor)

Circuit designers use m factors to mimic parallel copies of identical devices without having to instantiate large sets of devices in parallel. A design instance can inherit an m factor from one of its ancestors in a hierarchy of instances. The value of the inherited m factor in a particular module instance is the product of the m factor values in the ancestors of the instance and of the m factor value in the instance itself. The default value of an m factor is 1.0.

In the Cadence implementation of Verilog-A, you use the `inherited_mfactor` attribute to access the value of the m factor and set its value as follows:

```
(* inherited_mfactor *) parameter real m=1;
```

The following example illustrates how the m factor value is the product of the m factors in the current instance and in the ancestors of the current instance.

Consider a module declaration for `mf_res`, in a file called `mfactor_res.va`, in which you define the m factor to be one (`m=1`) using the `inherited_mfactor` attribute:

```
//
`include "discipline.vams"
`include "constants.vams"

module mf_res(vp, vn);
  inout vp, vn;
  electrical vp, vn;
  parameter real r=1;
  (* inherited_mfactor *) parameter real m=1;

  analog
    V(vp, vn) <+ r/m * I(vp, vn);

endmodule
```

Observe how we instantiate module `mf_res` in the following hierarchy and include the Verilog-A file that contains `mf_res` using an `ahdl_include` statement:

```
//
simulator lang=spectre

i1 (0 1) isource dc=1
r1 (0 1) my_sub_1 r=1k m=2
i2 (0 2) isource dc=1
r2 (0 2) my_sub_4 r=1k m=2

subckt my_sub_1(a b)
  parameters r=1
  ra (a b) mf_res r=r
ends my_sub_1

subckt my_sub_2(a b)
  parameters r=1
  ra (a b) my_sub_1 r=r m=2
ends my_sub_2
```

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

```
subckt my_sub_4(a b)
parameters r=1
ra (a b) my_sub_2 r=r m=2
ends my_sub_4

ahdl_include "mfactor_res.va"
```

```
save 1 2
mydc dc oppoint=screen
```

When we simulate this netlist, it generates results like the following, reflecting the division by  $m$  that appears in the `mf_res` module:  $V(vp, vn) <+ r/m * I(vp, vn);$

```
Instance: r1.ra of my_sub_1
Model: mf_res
Primitive: mf_res
    vp : val(0) = 0
    vn : V(1) = 500 V

Instance: r2.ra.ra.ra of my_sub_1
Model: mf_res
Primitive: mf_res
    vp : val(0) = 0
    vn : V(2) = 125 V
```

See also [“Setting an  \$m\$  Factor Directly on a Verilog-A Module”](#) on page 230.

## Setting an m Factor Directly on a Verilog-A Module

In the Cadence implementation of Verilog-A, you can set a multiplicity factor (m factor) directly on a Verilog-A module instance or on a subcircuit containing the Verilog-A module.

For example, you might include the Verilog-A file that contains the module using an `ahdl_include` statement in your netlist file, and set the value of `m` to be 100 on a Verilog-A module instance as follows:

```
...
ahdl_include "res.va"
r1 1 gnd res m=100
....
```

You do not need to declare or define the `m` factor in the Verilog-A module:

```
module res(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;
end

endmodule
```

The software scales instances by multiplying all current contributions by the value of `m` such that (for example):

```
V(a,b) <+ r*I(a,b);
```

becomes

```
V(a,b) <+ r*(I(a,b)/m);
```

which emulates `m` resistors in parallel.

Similarly, the software divides all current probes by the value of `m` such that (for example):

```
I(a,b) <+ 5u;
```

becomes

```
I(a,b) <+ m * 5u;
```

For contributions to a branch flow quantity using noise functions (such as `white_noise`, `flicker_noise`, and `noise_table`), the software multiplies the noise power by the value of `m`. For contributions to a branch potential quantity, the software divides the noise power by the value of `m`.

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

Multiplicity scaling also applies to flow contributions and flow probes of all disciplines; not just electrical.

Exclusions to this behavior are as follows:

- Modules that contain an inherited m factor
- Verilog-AMS modules

See also “Using the \$mfactor System Function” on page 232.

## Using the \$mfactor System Function

`$mfactor` is a Verilog-A hierarchical system parameter function that lets you access the shunt multiplicity factor (m factor) of a Verilog-A module instance. The m factor represents the number of identical instances you want to model as being in parallel. The value of an m factor in a particular module instance is the product of the m factor values in the ancestors of the instance, all the way to the top level, and of the m factor value in the instance itself. The default value of an m factor, when you do not specify one, is 1.0.

You can access the value of `$mfactor` in any module. You can use `$mfactor` to override the m factor for a child instance definition or use it in an `analog` block to change the behavior. You do not need to declare or define the m factor in the Verilog-A module.

**Note:** See also [“Overriding Parameter Values from the Module Instance Statement”](#) on page 222.

As with the [standard m factor](#), the following rules apply to `$mfactor` scaling:

- The software scales instances by multiplying all current contributions by the value of `$mfactor`
- The software divides all current probes by the value of `$mfactor`
- For contributions to a branch flow quantity using noise functions (such as `white_noise`, `flicker_noise`, and `noise_table`), the software multiplies the noise power by the value of `$mfactor`
- For contributions to a branch potential quantity, the software divides the noise power by the value of `$mfactor`

You can also use `$mfactor` to apply the m factor to other expressions.

See also

- [“\\$mfactor Double-Scaling”](#) on page 233
- [Using \\$mfactor Together with the Standard m Factor](#) on page 234
- [Using \\$mfactor Together with an Inherited m Factor](#) on page 235



## **\$mfactor Double-Scaling**

Verilog-AMS does not provide a method to disable the automatic \$mfactor scaling. The simulator will issue a warning if it detects a misuse of the \$mfactor in a manner that would result in double-scaling.

Examples:

The two resistor modules show the options of how \$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
```

**Note:** For more details on using \$mfactor, refer to *Verilog-AMS LRM*.

## Using \$mfactor Together with the Standard m Factor

If your design uses both m factor, `m`, and `$mfactor`, the calculation of the m factor in a particular module instance is still the product of the m factor values in the ancestors of the instance, all the way to the top level, and of the m factor value in the instance itself.

For example, perhaps you set the value of `m` somewhere in your netlist as follows:

```
...  
A1 1 2 module_A m=3  
...
```

and you have a module instance parameter override for `$mfactor` in a Verilog-A file such as:

```
module module_A (a, b)  
...  
module_b #(.$mfactor(2)) B1(p,n);  
endmodule  
  
module module_b (a, b)  
...  
endmodule
```

The effect in this case is that the m factor for instance `B1` is  $2 \times 3 \times m_{hierarchy}$ , where `mhierarchy` is the m-factor value that the software calculates by traversing the hierarchy from the top to the instantiating module.

The software traverses the hierarchy and replaces `$mfactor` with the standard m factor. Using the standard m factor, the software scales instances by multiplying all current contributions by the value of `m` and divides all current probes by the value of `m`.

**Note:** See “Setting an m Factor Directly on a Verilog-A Module” on page 230 for more information about automatic scaling using the standard m factor.

## Cadence Verilog-A Language Reference

### Instantiating Modules and Primitives

---

Notice how the following example changes according to this algorithm:

The following example:

```
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
```

becomes:

```
module parares(a,b);
inout a, b;
electrical a, b;

parameter real r = 1.0 from (0:inf);

analog begin
  if (r / m < 1.0e-3)
    V(a,b) <+ 0.0;
  else
    I(a,b) <+ V(a,b) / r * m;
end

endmodule
```

## Using \$mfactor Together with an Inherited m Factor

If you are using an inherited m factor in your design, you must not use \$mfactor, but instead use m directly. For example:

```
module parares(a,b);
inout a, b;
electrical a, b;

(* inherited_mfactor *) parameter real m = 1.0;
parameter real r = 1.0 from (0:inf);

analog begin
  if (r / m < 1.0e-3) // NOT if (r / $mfactor < 1.0e-3)
    V(a,b) <+ 0.0;
  else
    I(a,b) <+ V(a,b) / r * m;
end

endmodule
```

See also [“Setting an m Factor Directly on a Verilog-A Module”](#) on page 230.

# Cadence Verilog-A Language Reference

## Instantiating Modules and Primitives

---

---

## Controlling the Compiler

---

For information about controlling the Cadence® Verilog®-A compiler, see the following topics:

- [Using Compiler Directives](#) on page 238
- [Implementing Text Macros](#) on page 238
- [Compiling Code Conditionally](#) on page 240
- [Including Files at Compilation Time](#) on page 242
- [Setting Default Rise and Fall Times](#) on page 242
- [Resetting Directives to Default Values](#) on page 243

## Using Compiler Directives

The following compiler directives are available in Verilog-A. You can identify them by the initial accent grave ( ``` ) character, which is different from the single quote character ( `'` ).

- ``define`
- ``undef`
- ``ifdef`
- ``ifndef`
- ``include`
- ``resetall`
- ``default_transition`
- ``else`
- ``endif`

## Implementing Text Macros

By using the text macro substitution capability provided by the ``define` and ``undef` compiler directives, you can simplify your code and facilitate necessary changes. For example, you can use a text macro to represent a constant you use throughout your code. If you need to change the value of the constant, you can then change it in a single location.

### ``define` Compiler Directive

Use the ``define` compiler directive to create a macro for text substitution.

```
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 }
```

*macro\_text* is any text specified on the same line as *text\_macro\_name*. If *macro\_text* is more than a single line in length, precede each new-line character with a backslash ( `\` ). The first new-line character not preceded by a backslash ends *macro\_text*. You can include arguments from the *list\_of\_formal\_arguments* in *macro\_text*.

## Cadence Verilog-A Language Reference

### Controlling the Compiler

---

Subject to the restrictions in the next paragraph, you can include one-line comments in *macro\_text*. If you do, the comments do not become part of the text that is substituted. *macro\_text* can also be blank, in which case using the macro has no effect.

You must not split *macro\_text* across comments, numbers, strings, identifiers, keywords, or operators.

You can add `-va,define MACRO[=value]` to command line. This option defines a macro with priority higher than the one defined in Verilog-A files. Space is not allowed in the value.

*text\_macro\_identifier* is the name you want to assign to the macro. You refer to this name later when you refer to the macro. *text\_macro\_identifier* must not be the same as any of the compiler directive keywords but can be the same as an ordinary identifier. For example, `signal_name` and ``signal_name` are different.

#### *Important*

If your macro includes arguments, there must be no space between *text\_macro\_identifier* and the left parenthesis.

To use a macro you have created with the ``define` compiler directive, use this syntax:

```
text_macro_usage ::=
    `text_macro_identifier[( list_of_actual_arguments ) ]
list_of_actual_arguments ::=
    actual_argument { , actual_argument }
actual_argument ::=
    expression
```

*text\_macro\_identifier* is a name assigned to a macro by using the ``define` compiler directive. To refer to the name, precede it with the accent grave ( ``` ) character.

#### *Important*

If your macro includes arguments, there must be no space between *text\_macro\_identifier* and the left parenthesis.

`list_of_actual_arguments` corresponds with the list of formal arguments defined with the ``define` compiler directive. When you use the macro, each actual argument substitutes for the corresponding formal argument.

For example, the following code fragment defines a macro named `sum`:

```
`define sum(a,b) ((a)+(b)) // Defines the macro
```

To use `sum`, you might code something like this.

```
if (`sum(p,q) > 5) begin
    c = 0 ;
end
```

The next example defines an adc with a variable delay.

```
`define var_adc(dly) adc #(dly)
`var_adc(2) g121 (q21, n10, n11) ;
`var_adc(5) g122 (q22, n10, n11) ;
```

## **`undef Compiler Directive**

Use the ``undef` compiler directive to undefine a macro previously defined with the ``define` compiler directive.

```
undefine_compiler_directive ::=
    `undef text_macro_identifier
```

If you attempt to undefine a compiler directive that was not previously defined, the compiler issues a warning.

## **Compiling Code Conditionally**

Use the ``ifdef` or ``ifndef` compiler directive to control the inclusion or exclusion of code at compilation time. ef

### **`ifdef Compiler Directive**

```
conditional_compilation_directive ::=
    `ifdef text_macro_identifier
        first_group_of_lines
    [`else
        second_group_of_lines
    `endif ]
```

*text\_macro\_identifier* is a Verilog-A identifier. *first\_group\_of\_lines* and *second\_group\_of\_lines* are parts of your Verilog-A source description.

If you defined *text\_macro\_identifier* by using the ``define` directive, the compiler compiles *first\_group\_of\_lines* and ignores *second\_group\_of\_lines*. If you did not define *text\_macro\_identifier* but you include an ``else`, the compiler ignores *first\_group\_of\_lines* and compiles *second\_group\_of\_lines*.



## **`ifndef Compiler Directive**

```
conditional_compilation_directive ::=  
    `ifndef text_macro_identifier  
        first_group_of_lines  
    [`else  
        second_group_of_lines  
    `endif ]
```

`text_macro_identifier` is a Verilog-A identifier. `first_group_of_lines` and `second_group_of_lines` are parts of your Verilog-A source description.

If you did not define `text_macro_identifier` by using the ``define` directive, the compiler compiles `first_group_of_lines` and ignores `second_group_of_lines`. If you defined `text_macro_identifier` but you include an ``else`, the compiler ignores `first_group_of_lines` and compiles `second_group_of_lines`.

You can use the ``ifdef` or ``ifndef` compiler directive anywhere in your source description. You can, in fact, nest an ``ifdef` or ``ifndef` directive inside another ``ifdef` or ``ifndef` directive.

You must ensure that all your code, including code ignored by the compiler, follows the Verilog-A lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

## Including Files at Compilation Time

Use the ``include` compiler directive to insert the entire contents of a file into a source file during compilation.

```
include_compiler_directive ::=  
    `include "file"
```

*file* is the full or relative path of the file you want to include in the source file. *file* can contain additional ``include` directives. You can add a comment after the filename.

When you use the ``include` compiler directive, the result is as though the contents of the included source file appear in place of the directive. For example,

```
`include "parts/resistors/standard/count.va" // Include the counter.
```

would place the entire contents of file `count.va` in the source file at the place where the ``include` directive is coded.

Where the compiler looks for *file* depends on whether you specify an absolute path, a relative path, or a simple filename. If the compiler does not find the file, the compiler generates an error message.

## Setting Default Rise and Fall Times

Use the ``default_transition` compiler directive to specify default rise and fall times for the `transition` and `Z-transform` filters.

```
default_transition_compiler_directive ::=  
    `default_transition transition_time
```

*transition\_time* is an integer value that specifies the default rise and fall times for `transition` and `Z-transform` filters that do not have specified rise and fall times.

If your description includes more than one ``default_transition` directive, the effective rise and fall times are derived from the immediately preceding directive.

The ``default_transition` directive sets the transition time in the `transition` and `Z-transform` filters when local transition settings are not provided. If you do not include a ``default_transition` directive in your description, the default rise and fall times for `transition` and `Z-transfer` filters is 0.

## Resetting Directives to Default Values

Use the ``resetall` compiler directive to set all compiler directives, except the ``timescale` directive, to their default values.

```
resetall_compiler_directive ::=  
    `resetall
```

Placing the ``resetall` compiler directive at the beginning of each of your source text files, followed immediately by the directives you want to use in that file, ensures that only desired directives are active.

**Note:** Use the ``resetall` directive with care because it resets the

```
`define DISCIPLINES_VAMS
```

directive in the `discipline.vams` file, which is included by most Verilog-A files.

**Cadence Verilog-A Language Reference**  
Controlling the Compiler

---

---

## AHDL Linter Checks

---

This chapter contains the following topics:

- [About the AHDL Linter Feature](#) on page 246
- [Using the AHDL Linter Feature in APS, XPS and UltraSim](#) on page 246
- [Identifying AHDL Linter Messages](#) on page 248
- [Filtering AHDL Linter Messages](#) on page 249
- [Using the ahdlhelp Utility](#) on page 249

## About the AHDL Linter Feature

The AHDL Linter technology offers a powerful set of capabilities for upfront identification of issues that worry designers even well beyond the model creation stage. The AHDL Linter feature helps identify complex design modeling issues and can be used on block simulation or SOC verifications performed just before tape out. Early warnings enable designers to avoid expensive design iterations, and meeting quality and time-to-market goals.

You can use the AHDL Linter feature to analyze the APS, XPS, and UltraSim test cases containing Verilog-A models.

AHDL Linter checks each Verilog-A behavioral model in the design and suggests which line should be improved to:

- Avoid potential convergence or performance problems
- Improve model accuracy, reusability, and portability

AHDL linter consists of static and dynamic lint checks. Static lint checks are performed at compilation stage. Dynamic lint checks are performed during analysis for dynamic modeling issues that may cause convergence or performance problems during simulation.

## Using the AHDL Linter Feature in APS, XPS and UltraSim

The AHDL Linter feature is activated in APS, XPS, and UltraSim using the following command-line options:

```
% spectre +aps -ahdllint netlist.scs
% spectre +xps +ckptpreset=sram -ahdllint netlist.scs
% ultrasim -ahdllint netlist.scs
```

where `netlist.scs` is written using Spectre, SPICE, or eSpice syntax and includes one or more Verilog-A models.

You can perform static linter checks on a netlist file that includes one or more Verilog-A models, as follows:

```
spectre +aps -ahdllint=static test.scs // test.scs includes the Verilog-A files
```

Spectre, in this case, performs only linter checks on all Verilog-A files and does not run the simulation.

You can also perform static linter checks on a Verilog-A file directly without specifying the top-level netlist file (`.scs`), as follows:

```
spectre +aps -ahdllint=static test.va //perform static linter check on test.va
```

## Cadence Verilog-A Language Reference

### AHDL Linter Checks

---

**Note:** Ultrasim does not support static linter check for a single Verilog-A file.

The `-ahdllint` option accepts the following arguments:

<code>no</code>	Disables lint checks. There is no change in the existing compilation or simulation error messages.
<code>warn</code>	(default) Turns on the static lint and dynamic lint checks. Except the models with attribute ( <code>* ahdllint=no *</code> ), the static linter checks all models, continues the simulation, and then performs dynamic lint checks.
<code>error</code>	Turns on the static lint check. Dynamic lint checks are performed only when static lint issues are not detected. Except the models with attribute ( <code>* ahdllint=no *</code> ), the static linter checks all models. The simulator generates an error and exits if there is any static lint warning reported after compiling all the models of the circuit. If there are no static lint warnings, the simulator continues the simulation and performs dynamic lint checks. However, in the case of dynamic lint issues, the simulator does not error out.
<code>force</code>	Similar to <code>warn</code> , but this option overrides the model attribute ( <code>* ahdllint=no *</code> ), and forces to check all models, continue the simulation, and perform dynamic lint checks.
<code>static</code>	Performs static linter checks on a Verilog-A file directly.

**Note:** The `-ahdllint` option can be used without any argument, which is equivalent to specifying `-ahdllint=warn`. You can set the `-ahdllint` option as default by setting the `SPECTRE_DEFAULTS` environment variable, as follows:

```
setenv SPECTRE_DEFAULTS "-ahdllint=warn"
```

You can use the `-ahdllint_maxwarn=n` command-line option to control the maximum number of static and dynamic warnings. The default value is 5.

You can also control the maximum number of warnings by using the environment variable `SPECTRE_DEFAULTS` or `ULTRASIM_DEFAULTS`, as shown below.

```
setenv SPECTRE_DEFAULTS "-ahdllint -ahdllint_maxwarn=10"
setenv ULTRASIM_DEFAULTS "-ahdllint -ahdllint_maxwarn=10"
```

You can use the `-ahdllint_minstep=value` command-line option to set the minimal time step size boundary that will trigger an AHDL Linter warning, when the step size imposed by a Verilog-AMS filter or function, such as `transition`, `cross`, `above`, `$bound_step`, and `timer` is smaller than `value`. The default value is `1e-12`.

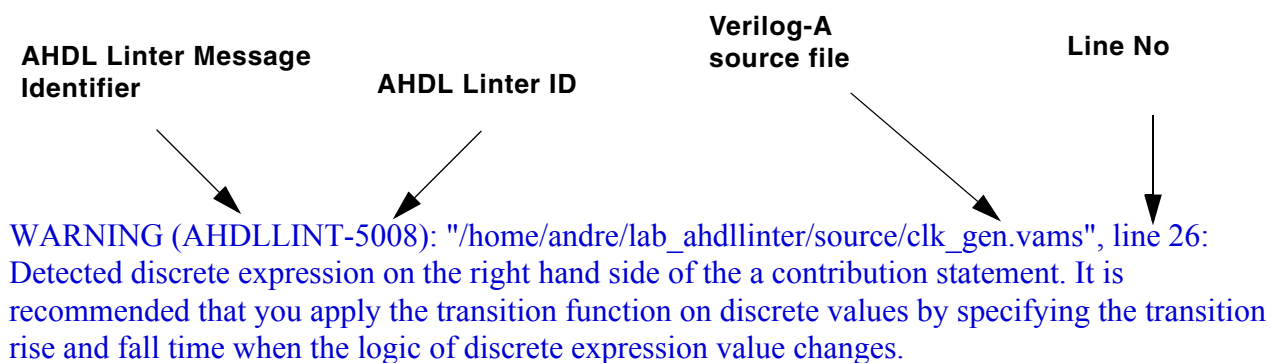
By default, the static and dynamic linter warnings and the dynamic linter summary output are sent to the simulation log file. You can use the `-ahdllint_log=file` command-line option to output all the information to a specified file instead of the output log file.

## Identifying AHDL Linter Messages

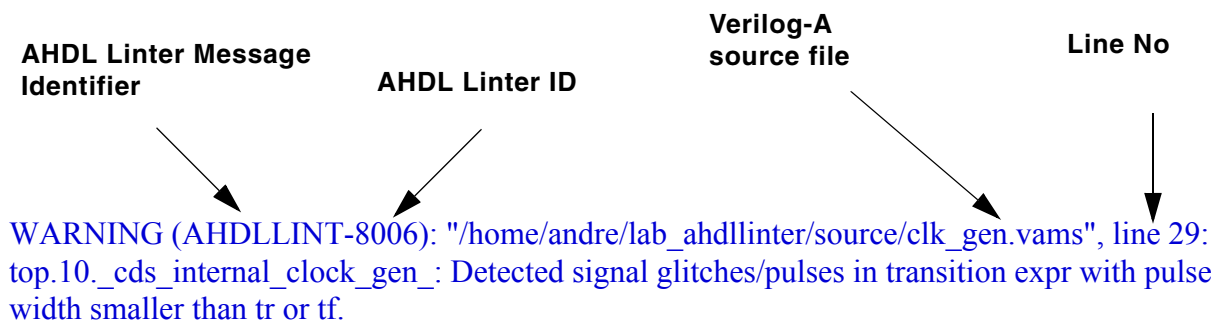
The messages generated by AHDL Linter (static or dynamic) have the message identifier `AHDL LINT` at the beginning of the message followed by the message ID.

The following examples show the static and dynamic AHDL linter messages generated by APS, XPS, and UltraSim:

### Static AHDL Linter Message



### Dynamic AHDL Linter Message





## Filtering AHDLLinter Messages

You can filter the AHDLLinter messages using the `options` control statement, as follows:

```
Name options warning_limit=num warning_id=id
```

where:

`warning_limit` specifies the number of messages to be printed.

`warning_id` specifies the message ID that needs to be printed.

### Example1

```
myoptions options warning_limit=0 warning_id=[AHDLLINT-8004 AHDLLINT-8005]
```

In the above example, APS, XPS, and UltraSim will not print any message with the ID AHDLLINT-8004 and AHDLLINT-8005.

### Example2

```
myoptions2 options warning_limit=1 warning_id=[AHDLLINT-8005]
```

In the above example, APS, XPS, and UltraSim will print only one message with the ID AHDLLINT-8005.

## Using the `ahdlhelp` Utility

You can use the `ahdlhelp` utility to view the extended help on various messages reported by AHDLLinter. To view the extended help, you need to provide the AHDLLinter ID number as an argument, as shown below.

```
% ahdlhelp 8005
```

The following is an example of the `ahdlhelp` utility:

```
% ahdlhelp 5012
```

```
AHDLLINT-5012: Detected $abstime in an equality expression inside a conditional statement.
```

```
Analog simulations select timepoints using a variable timestep size algorithm based on accuracy considerations, so the value of "time" only changes between discrete real values. In order to perform an event at a particular time, the @(timer) construct must be used to tell the simulator to step to a particular timepoint and execute the desired code when that event actually occurs.
```

```
example:
```

```
    if ($abstime==50n) xval=2;
```

```
solution:
```

```
    @(timer(50n)) xval=2;
```

# Cadence Verilog-A Language Reference

## AHDL Linter Checks

---

---

## Using Verilog-A in the Cadence Analog Design Environment

---

This chapter describes how to use Cadence® Verilog®-A in the Cadence analog design environment.

You must use the Spectre® circuit simulator or the SpectreVerilog circuit simulator—with the spectre or spectreVerilog interface—to simulate designs that include Verilog-A components.

This chapter discusses:

- [Creating Cellviews Using the Cadence Analog Design Environment](#) on page 252
- [Using Escaped Names in the Cadence Analog Design Environment](#) on page 264
- [Defining Quantities](#) on page 264
- [Using Multiple Cellviews for Instances](#) on page 266
- [Multilevel Hierarchical Designs](#) on page 285
- [Using Models with Verilog-A](#) on page 290
- [Saving Verilog-A Variables](#) on page 291
- [Displaying the Waveforms of Variables](#) on page 291

**Note:** When you run the Verilog-A language in the analog design environment, there are a few differences from running the Verilog-A language standalone:

- Always use a full path when opening files inside a module using `$fopen`. Reading and writing files can be a problem if you do not use a full path. The analog design environment might use a run directory that is in a different location than what you expect.
- Code in the Verilog-A language that relies on command line arguments or environment variables might cause a problem because the analog design environment controls or limits certain command line options.

- When you are using the analog design environment, editing the Verilog-A source files might cause a problem. For more information, see [“Editing Verilog-A Cellviews Outside of the Analog Design Environment”](#) on page 258.

## Creating Cellviews Using the Cadence Analog Design Environment

This section describes how to create symbol, block, and Verilog-A cellviews in the analog design environment.

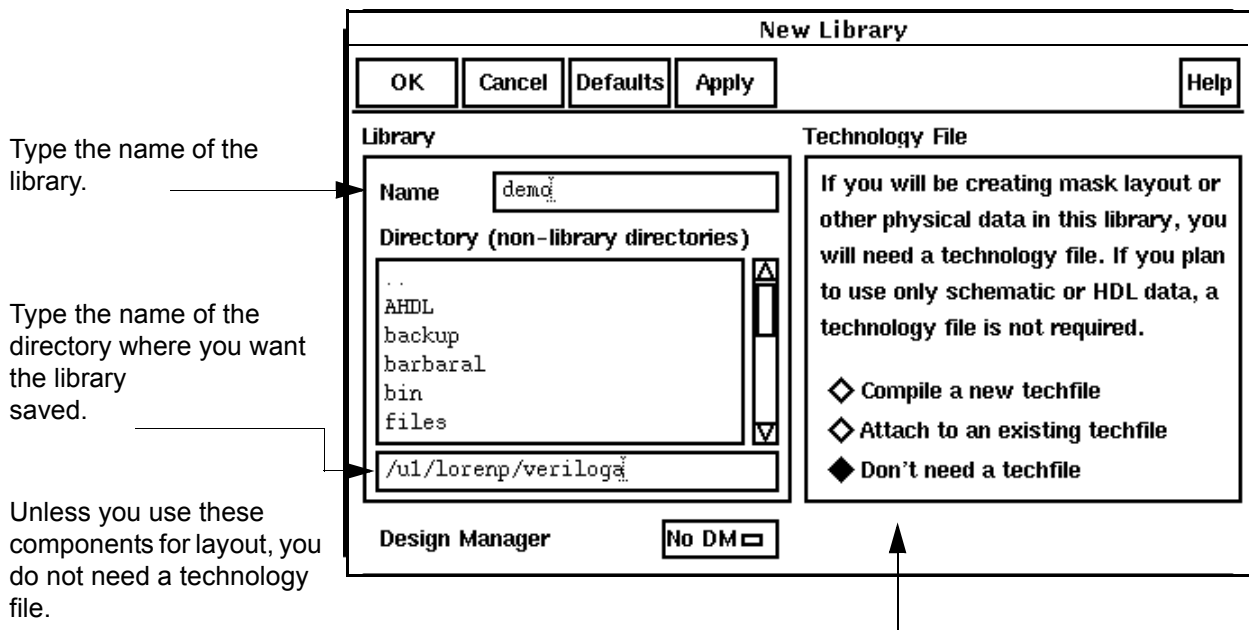
### Preparing a Library

Before you create a cell, you must have a library in which to place it. You can create and store Verilog-A components in any Cadence component library. You can create a new library or use one that already exists.

To create a new library, follow these steps:

1. In the Command Interpreter Window (CIW), choose *File – New – Library*.

The New Library form opens.



## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

2. In the New Library form, type the new library name and directory and click on the radio button for no techfile. Click *OK*.

A message appears in the CIW:

```
Created library "library_name" as "dir_path/library_name"
```

The *library\_name* and *dir\_path* are the values that you specified.

You can also use the Cadence library manager to create a new library.

1. In the CIW, choose *Tools – Library Manager*.

The library manager opens.

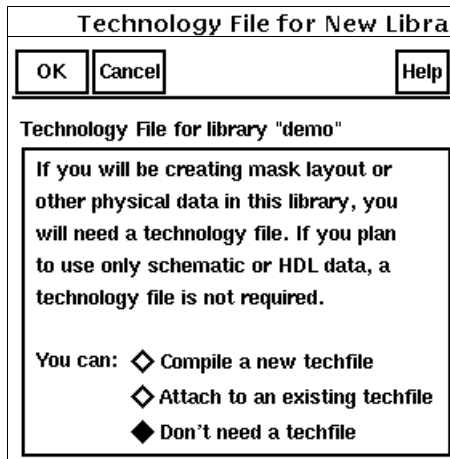
2. Choose *File – New – Library*.

The New Library form opens. This form is different from the New Library form that you can open from the CIW.

The screenshot shows a dialog box titled "New Library". It is divided into three main sections. The top section, "Library", contains a "Name" text field with the text "demo" and a "Directory" list box. The list box contains the following items: "mydoc", "nsmail", "opus", "osc", "simulation", and "status". Below the list box is a text field containing the path "/u1/lorenp/verilogs". The middle section, "Design Manager", contains two radio buttons: "Use NONE" and "Use No DM". The bottom section contains four buttons: "OK", "Apply", "Cancel", and "Help".

3. In the *Name* field, type the new library name.
4. In the *Directory* list box, choose the directory where you want to place the library.
5. Click *OK*.

A second form opens, asking if you need a technology file for this library.



6. Set *Don't need a techfile* on and click *OK*.

The analog design environment creates a new library with the name you specify in the directory you specify. The following appears in the CIW display area:

```
Library Manager created library "library_name".
```

## Creating the Symbol View

To include a Verilog-A module in a schematic, you must create a symbol to represent the function described by the module. There are four ways to create this symbol:

- Choose *File – New – Cellview* from the CIW and specify the target application as *Composer-Symbol*.
- Copy an existing symbol using the *Copy* command in the library manager. Look in *analogLib* for good examples to copy.
- Create a new symbol from another view using *Design – Create Cellview – From Pin List* or *Design – Create Cellview – From Cellview* in the Schematic Design Editor. To create a new symbol this way, you must first have an existing view with defined input and output pins.
- Use a block to represent a Verilog-A function, as described in [“Using Blocks”](#) on page 255.

However you create the symbol, it must reside in an existing library as described in [“Preparing a Library”](#) on page 252.

## Pin Direction

The direction you assign to a symbol pin (Verilog-A defines pin direction) does not affect that terminal in the Verilog-A module. However, if you have multiple cellviews for a component, make sure that the name (which can be mapped), type, and location of pins you assign in a symbol cellview match what is specified in the other cellviews.

## Buses

Verilog-A modules support vector nodes and branches, also known as buses or arrays. For more information about declaring buses in Verilog-A modules, see [“Net Disciplines”](#) on page 75.

## Using Blocks

In top-down design practice, you can use blocks to represent Verilog-A functions. You can create blocks at any level in your design, even before you know how the individual component symbols should look.

In a schematic, to create a block and wire it, follow these steps:

1. Choose *Add – Block* in the Virtuoso Schematic Editing window.

The Add Block form opens.

Add Block	
Hide	Cancel
Defaults	Help
Library	demo
Cells	vdba
View	symbol
Names	
Pin Name Prefix	pin
Block Shape	medium <input type="checkbox"/>

2. Type a library name, cell name, and view name.

Specify a cell and view combination that does not exist in that library. You can have schematic or Verilog-A views for that cell, but you cannot already have a symbol view. The default library name is the current library, and the default view name is `symbol`.

3. (Optional) Specify the pin name seed to use when you connect a wire to the block.

If you specify a seed of `pin`, the schematic editor names the first pin that you add `pin1`, names the second pin `pin2`, and so on.

4. Set the *Block Shape* cyclic field.
5. Place the block as described in the following table.

<b>If Block Shape is set to freeform</b>	<b>If Block Shape is set to anything else</b>
Press the left mouse button where you want to place the first corner of the rectangle and drag to the opposite corner. Release the mouse button to complete the block.	Drag the predefined block to the location where you want to place it and click.  Refer to the <i>Virtuoso Schematic Editor User Guide</i> for details about modifying the block samples using the <code>schBlockTemplate</code> variable in the <code>schConfig.il</code> file.

As you place each block, the schematic editor labels it with an instance name. If you leave the *Names* field of the Add Block form empty, the editor generates unique new names for the blocks.

The editor automatically creates a symbol view for the block.

6. Choose *Add – Wire (narrow)* or *Add – Wire (wide)* from the Virtuoso® schematic composer window menu. When you connect the wire, the pin is created automatically. (To delete such a pin, you must use *Design – Hierarchy – Descend Edit* to descend into the block symbol.)

The *Pin Name Prefix* field on the Add Block form specifies the name for the automatically created pin.

## SKILL Function

Use this Cadence SKILL language function to create a block instance:

```
schHiCreateBlockInst
```

## Creating a Verilog-A Cellview from a Symbol or Block

Once you have an existing symbol or block, you can create an Verilog-A cellview for the function identified by that symbol or block. To create the cellview, follow these steps:

1. Open the Symbol Editor in one of two ways:



## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

- ❑ In the CIW, choose *File – Open* and specify the component or block symbol.
  - ❑ In the library manager, choose *File – Open* or double-click on the symbol view.
2. In the Symbol Editor window, choose *Design – Create Cellview – From Cellview*.  
The Cellview From Cellview form opens.

The screenshot shows the 'Cellview From Cellview' dialog box. The title bar reads 'Cellview From Cellview'. Below the title bar are five buttons: 'OK', 'Cancel', 'Defaults', 'Apply', and 'Help'. The main area contains several input fields and a 'Browse' button. The 'Library Name' field contains 'interface'. The 'Cell Name' field contains 'vdba'. The 'From View Name' field contains 'symbol' and has a small square icon to its right. The 'To View Name' field contains 'veriloga'. The 'Tool / Data Type' field contains 'VerilogA-Editor' and has a small square icon to its right. At the bottom of the dialog, there are two checkboxes: 'Display Cellview' and 'Edit Options', both of which are checked.

3. In the *From View Name* cyclic field, choose *symbol*; in the *Tool / Data Type* cyclic list, choose *VerilogA-Editor*; and, in the *To View Name* field, type *veriloga*. The view name *veriloga* is the default view name for Verilog-A views.

When you click *OK*, an active text editor window opens, showing the template for a Verilog-A module.

```
//VerilogA for demo, vdba, veriloga
`include "constants.vams"
`include "disciplines.vams"

module vdba(out, in);
output out;
electrical out;
input in;
electrical in;
parameter real gain = 0.0 ;
parameter real vin_high = 0.0 ;
parameter real vin_low = 0.0 ;

endmodule
```

The analog design environment creates the first few lines of the module based on the symbol information. Pin and parameter information are included automatically, but you might need to edit this information so that it complies with the rules of the Verilog-A language.

4. Finish coding the module, then save the file and quit the text editor window. The analog design environment does not create the cellview until you exit from the editor.

Here is an example of a completed module:

```
//VerilogA for demo, vdba, veriloga
`include "constants.vams"
`include "disciplines.vams"
module vdba(out, in);
output out;
electrical out;
input in;
electrical in;
parameter real vin_low = -2.0 ;
parameter real vin_high = 2.0 ;
parameter real gain = 1 from (0:inf) ;
analog begin
    if (V(in) >= vin_high) begin
        V(out) <+ gain*(V(in) - vin_high) ;
    end else if (V(in) <= vin_low) begin
        V(out) <+ gain*(V(in) - vin_low) ;
    end else begin
        V(out) <+ 0 ;
    end
end
```

When you save the module and quit the text editor window, the analog design environment checks the syntax in the text file. If the syntax checker finds any errors or problems, a dialog box opens with the following message.

```
Parsing of analog_hdl file failed:
Do you want to view the error file and re-edit the analog_hdl file?
```

Click **Yes** to display the *analog\_hdl* Parser Error/Warnings window and to reopen the module file for editing.

If the syntax checker does not find any errors or problems, you get this message in the CIW:

```
analog_hdl Diagnostics: Successful syntax check for analog_hdl text of cell
cellname.
```

## Editing Verilog-A Cellviews Outside of the Analog Design Environment

The analog design environment parses the Verilog-A code after the module is saved and then uses this information as the basis for creating the netlist.

Do not directly edit the source files if you need to change the module name, cell name, parameter names, parameter values, pin names, or the body of a module or if you need to add or delete pins or parameters. Instead, use the analog design environment for these changes. When you use the analog design environment, the parser communicates hierarchical element information to the netlister to automatically include other Verilog-A

## **Cadence Verilog-A Language Reference**

### Using Verilog-A in the Cadence Analog Design Environment

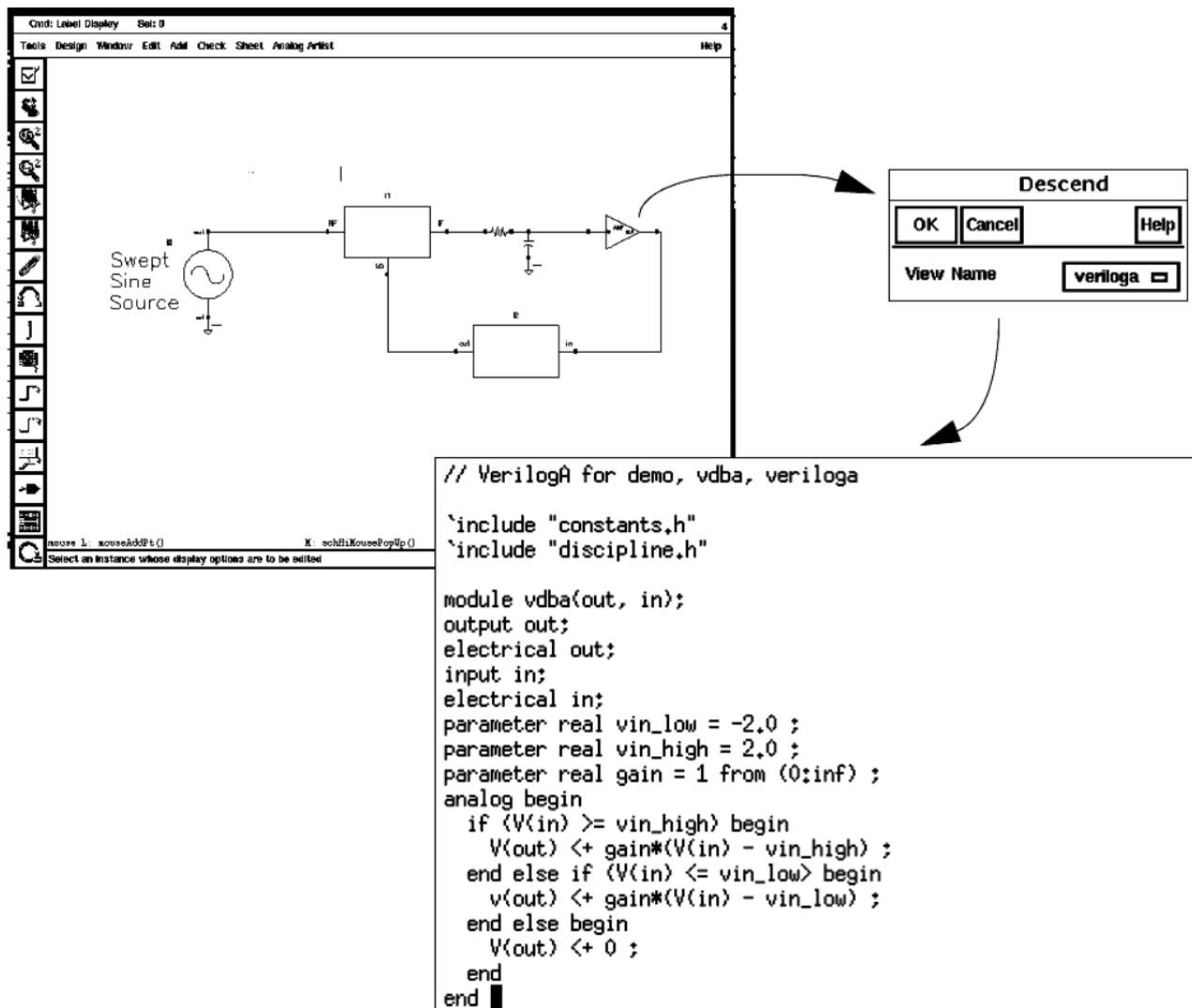
---

module definitions in the final netlist. When you edit directly, however, the parser does not run and cannot send the required hierarchical information to the netlister.

If you change a file that is included (with a `#include` statement) in a Verilog-A module, you must then re-edit or recompile the Verilog-A module in the analog design environment. If you change the included file without re-editing or recompiling the compiled information, the compiled information for the Verilog-A module might not match the actual module definition. This inconsistency results in an incorrect netlist.

## Descend Edit

To examine the views below the symbols while viewing a schematic, choose *Design – Hierarchy – Descend Edit*. For example, there might be two view choices: *symbol* and *veriloga*. If you choose *veriloga*, a text window opens, as shown in the following figure.



The figure shows a screenshot of the Cadence Analog Design Environment. On the left, a schematic diagram is displayed, featuring a 'Swept Sine Source' connected to a network of components including a block labeled 'n', a resistor, a capacitor, and an op-amp. A 'Descend' dialog box is open over the schematic, with 'veriloga' selected in the 'View Name' field. Below the schematic, a text window displays the Verilog-A code for a component named 'vdba'.

```
// VerilogA for demo, vdba, veriloga
`include "constants.h"
`include "discipline.h"

module vdba(out, in);
output out;
electrical out;
input in;
electrical in;
parameter real vin_low = -2.0 ;
parameter real vin_high = 2.0 ;
parameter real gain = 1 from (0:inf) ;
analog begin
  if (V(in) >= vin_high) begin
    V(out) <+ gain*(V(in) - vin_high) ;
  end else if (V(in) <= vin_low) begin
    v(out) <+ gain*(V(in) - vin_low) ;
  end else begin
    V(out) <+ 0 ;
  end
end
end
```

## Creating a Verilog-A Cellview

To create a new component with only a Verilog-A cellview, follow these steps:

1. In the CIW, choose *File – New – Cellview*.

The Create New File form opens.

**Create New File**

OK Cancel Defaults Help

Library Name

Cell Name

View Name

Tool

Library path file

2. Specify the *Cell Name* (component).
3. Specify the view that you want to create.

To create a new veriloga view, set the *Tool* cyclic field to *VerilogA-Editor*.

4. In the *View Name* field, type the name for the new cellview.
5. Click *OK*.

A text editor window opens for the new module. If the cell name you typed in the *Cell Name* field is new, an empty template opens. If the name you typed already has available views, a template opens with pin and parameter information in place.

By default, the module has the same name as the cell. →

The UNIX file directory has the same name as the cellview. →

/u1/lorenp/Demo/AHDL/vdba/veriloga/veriloga.va

```

VerilogA for AHDL, vdba, veriloga
`include "constants.h"
`include "discipline.h"
module vdba;
endmodule
~
~
        
```

6. Modify any existing pin or parameter information as necessary. You can add unique or shared parameters as required by your design.
7. If you want to simulate multiple views of a cell at the same time, change the new module name so that it is unique for each view.
8. Complete the module, save it, and quit the text editor window.

### Creating a Symbol Cellview from a New Analog HDL Cellview

After you save and quit a newly created Verilog-A file, a dialog box opens. It tells you that no symbol exists for this cell and asks you if you want to create a symbol. To create a symbol, follow these steps:

1. Click Yes.

The Symbol Generation Options form opens.

Symbol Generation Options	
OK	Cancel
Apply	Help
Library Name	Cell Name
View Name	
interface	vdb
	symbol
Pin Specifications	
Left Pins	Attributes
in	List
Right Pins	List
out	List
Top Pins	List
	List
Bottom Pins	List
	List
Load/Save	Edit Attributes
Edit Labels	Edit Properties

2. Edit the pin information for your symbol as required.
3. Set *Load/Save* on.
4. Click *OK*.

The Symbol Generation Options form closes, and the Symbol Editor form opens. Any warnings appear in the CIW.

If you receive any warnings, take time to check the symbol and examine the Component Description Format (CDF) information for your new cell.

5. Edit the symbol and save it.
6. Close the Symbol Editor form.

## Creating a Symbol Cellview from an Analog HDL Cellview

If you created a Verilog-A cellview without creating a symbol, or if you have a component with only a Verilog-A cellview, you can add a symbol view to that component. The easiest way to add a symbol view is to reopen the Verilog-A cellview, write the information, and close the cellview. When you are asked if you want to create a symbol for the component, click **Yes** and follow the procedure in [“Creating a Symbol Cellview from a New Analog HDL Cellview”](#) on page 262.

You can also add a symbol view by following these steps:

1. Choose *File – Open* from the CIW.

The Open File form opens.

2. Open any schematic or symbol cellview.

The editor opens.

3. Choose *Design – Create Cellview – From Cellview*.

The Cellview From Cellview form opens.

Cellview From Cellview				
<input type="button" value="OK"/>	<input type="button" value="Cancel"/>	<input type="button" value="Defaults"/>	<input type="button" value="Apply"/>	<input type="button" value="Help"/>
Library Name	<input type="text" value="demo"/>	<input type="button" value="Browse"/>		
Cell Name	<input type="text" value="vdba"/>			
From View Name	<input type="text" value="veriloga"/>	To View Name	<input type="text" value="symbol"/>	
		Tool / Data Type	<input type="text" value="Composer-Symbol"/>	
Display Cellview	<input checked="" type="checkbox"/>			
Edit Options	<input checked="" type="checkbox"/>			

4. Fill in the *Library Name* and *Cell Name* fields.

If you do not know this information, click *Browse*, which opens the Library Browser, so you can browse available libraries and components.

5. In the *From View Name* cyclic field, select the Verilog-A view.

6. In the *Tool / Data Type* cyclic field, choose *Composer-Symbol*.
7. In the *To View Name* field, type `symbol`.
8. Click *OK*.  
The Symbol Generation Options form opens.
9. Click *OK*.  
A Symbol Editor window opens.
10. Edit the symbol, save it, and close the Symbol Editor window.

## Using Escaped Names in the Cadence Analog Design Environment

As described in “[Escaped Names](#)” on page 49, the Verilog-A language permits the use of escaped names. The analog design environment, however, does not recognize such names. As a consequence,

- You must not use escaped names for modules that the analog design environment instantiates directly in a netlist, nor can you use escaped names for the parameters of such modules
- Although you can use escaped names for formal module ports, you cannot use escaped names in the corresponding actual ports of module instances instantiated in the netlist

## Defining Quantities

To use a custom quantity in a Verilog-A module, you can define the quantity in a Spectre netlist or in a Verilog-A discipline. A quantity defined in a netlist overrides any definition for that quantity located in a Verilog-A discipline. See the [Spectre Circuit Simulator User Guide](#) for more information.

You need to place a file named `quantity.spectre` in libraries you create that contain Verilog-A or modules that use custom quantities. `quantity.spectre` specifies these custom quantities and their default values. When generating the netlist, the Cadence analog design environment searches each library in your library search path for `quantity.spectre` files and then automatically adds `include` statements for these files into the netlist.

The format of the `quantity` statement is defined by the [Spectre quantity component](#) (see `spectre -h quantity` or the [Virtuoso Spectre Circuit Simulator Reference](#) manual).



## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

```
quantity_statement ::=  
    instance_name quantity { parameter=value }
```

*instance\_name* is the reference for this line in the netlist. You must ensure that *instance\_name* is unique in the netlist.

*parameter* is one of the parameters listed in the following table. The corresponding *value* must be of the appropriate type for each parameter. To specify a list of parameters, separate them with spaces.

#### Quantity Parameters

---

Parameters	Required or Optional?	The value must be
abstol	Required	A real value
blowup	Optional	A real value
description	Optional	A string
huge	Optional	A real value
name	Required	A string
units	Optional	A string

---

For example, a `quantity.spectre` file might contain information such as the following:

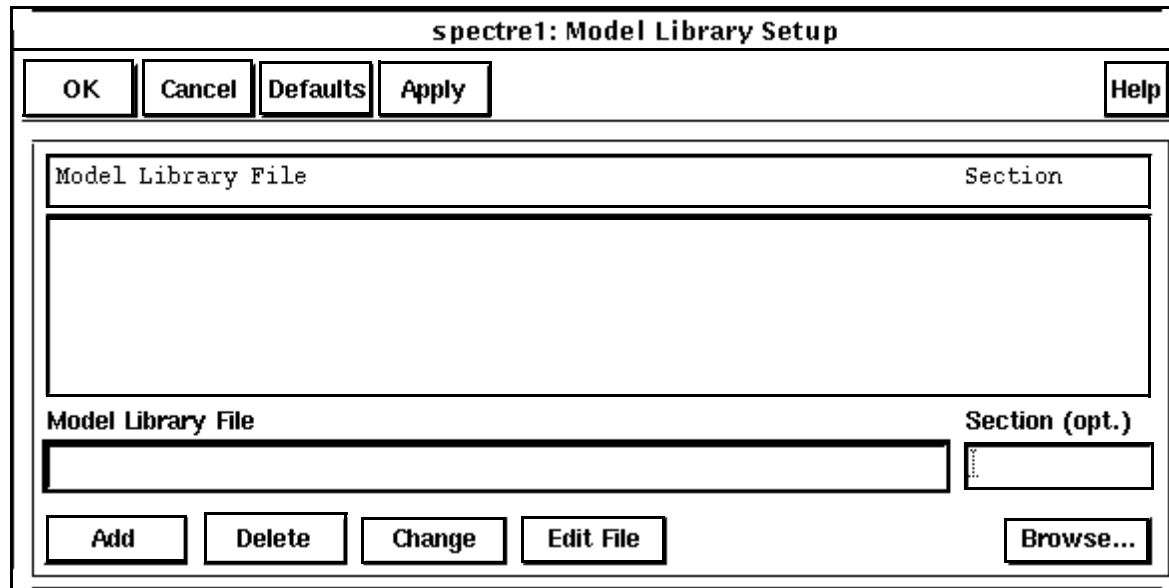
```
displacementX quantity name="X" units="M" abstol=1m  
displacementY quantity name="Y" units="ft" abstol=1m  
torque quantity name="T" units="N" abstol=1m blowup=1e9  
omega quantity name="W" units="rad/sec" abstol=1m
```

**Note:** Each quantity must have a unique `name` parameter to identify it. You can redefine the parameters for a specific quantity by using a new `quantity` statement in which the `name` parameter is the same and the other parameters are set as required.

#### spectre/spectreVerilog Interface (Spectre Direct)

To override values set by a `quantity.spectre` file or to insert a specific set of quantities into a module, you can specify the UNIX path of a file that contains `quantity` statements in the Model Library Setup form. Cadence recommends that you use the full path.

**Note:** If you do use relative paths, be aware that they are relative to the netlist directory, not the `icms` run directory.



**Note:** The `ahdlIncludeFirst` environment variable is not used for the `spectre` and `spectreVerilog` interfaces and is ignored by them.

## Using Multiple Cellviews for Instances

As you develop a design, you might find it useful to have more than one veriloga cellview for a given instance of a component. For example, you might want to have two or more veriloga cellviews with different behaviors and parameters so that you can determine which works best in your design. The next few sections explain how to use the multiple Verilog-A cellview capability that is built into the Cadence analog design environment.

Designs created before product version 4.4.2 must be updated before you can use the multiple analog HDL cellview capability. Cadence® provides the `ahdlUpdateViewInfo` SKILL function that you can use to update your design.

For the greatest amount of compatibility with Cadence AMS Designer, Cadence recommends that each module have the same name as the associated cell. (However, this approach is not supported for hierarchies of Verilog-A modules.)

For example, assume that you want to be able to switch between two veriloga definitions of the cell `ahdlTest`. One of the definitions, which is assumed to have the view name `verilogaone`, is defined by the module

## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

```
module ahdlTest(a)
    electrical a ;
    analog
        V(a) <+ 10.5 ;
endmodule
```

The other veriloga definition, which has the view name `verilogatwo`, is defined by the module

```
module ahdlTest(a)
    electrical a ;
    analog
        V(a) <+ 9.5 ;
endmodule
```

Now, assuming that all the cells are stored in the library `myAMSLib`, these views are referred to as `myAMSLib.ahdlTest:verilogaone` and `myAMSLib.ahdlTest:verilogatwo`. To switch from one version of the cell to the other, you can then use the Cadence hierarchy editor, for example, to bind the view that you want to use.

## Creating Multiple Cellviews for a Component

You can create as many Verilog-A cellviews for a component as you need. You can give a new cellview any name except the name of an existing cellview for the component. Whatever you name a new cellview, its view type is determined by the application you use to create the new cellview. As described earlier in this chapter, you can create new Verilog-A cellviews, from symbols, and from blocks. You can also create new Verilog-A cellviews from existing analog HDL cellviews.

## Creating Verilog-A Cellviews from Existing Verilog-A Cellviews

To create a Verilog-A cellview from an existing Verilog-A cellview, follow these steps:

1. Choose *File – Open* from the CIW.  
The Open File form opens.
2. Open any schematic or symbol cellview.  
The editor opens.
3. Choose *Design – Create Cellview – From Cellview*.

The Cellview From Cellview form opens.

The screenshot shows the 'Cellview From Cellview' dialog box. At the top, there are buttons for 'OK', 'Cancel', 'Defaults', 'Apply', and 'Help'. Below these are several input fields: 'Library Name' (text: 'demo'), 'Cell Name' (text: 'vdbal'), 'From View Name' (dropdown menu: 'veriloga'), 'To View Name' (text: 'ahdl'), and 'Tool / Data Type' (dropdown menu: 'SpectreHDL-Editor'). There is also a 'Browse' button next to the 'Library Name' field. At the bottom, there are two checkboxes: 'Display Cellview' and 'Edit Options', both of which are checked.

4. Fill in the *Library Name* and *Cell Name* fields with information for the existing cellview.

If you do not know this information, click *Browse* to see the available libraries and components.

5. In the *From View Name* cyclic field, choose the existing cellview.
6. In the *Tool /Data Type* cyclic field, choose the application that creates the type of cellview you want.
7. If necessary, edit the cellview name that appears in the *To View Name* field.
8. Click *OK*.

A template opens.

9. Complete the module, save it, and quit the text editor window.

## Modifying the Parameters Specified in Modules

By default, instances of Verilog-A components use the parameter values in their defining text modules. However, if you want different parameter values, you can use the Edit Object Properties form in the Virtuoso® schematic composer to individually modify the values for each cellview available for the instance. You can change parameter values for the cellview currently bound with an instance, and you can change the parameter values of cellviews that are available for an instance but not currently bound with it.

To take full advantage of multiple cellviews, your schematic must be associated with a configuration. If you do not have a configuration, you need to create one. For guidance, see the *Cadence Hierarchy Editor User Guide*.

### Opening a Configuration and Associated Schematic

To open a configuration and its associated schematic, follow these steps:

1. In the library manager, highlight the config view for the cell you want to open.
2. Choose *File – Open*.

The Open Configuration or Top CellView form opens.

Open Configuration or Top CellView		
OK	Cancel	Help
Open for editing		
Configuration "AHDL demogain config"	<input checked="" type="radio"/> yes	<input type="radio"/> no
Top Cell View "AHDL demogain schematic"	<input checked="" type="radio"/> yes	<input type="radio"/> no

3. Select yes to open the configuration and yes to open the top cell view.
4. Click *OK*.

The Cadence hierarchy editor and Virtuoso Schematic Editing windows both open.

### Changing the Parameters of a Cellview Bound with an Instance

To change the parameter values of a cellview bound with an instance, follow these steps:

1. Select the instance in the Virtuoso Schematic Editing window.
2. Choose *Edit – Properties – Objects*.

The Edit Object Properties form opens.

Property	Value	Display
Library Name	AHDL	off
Cell Name	gain	off
View Name	symbol	off
Instance Name	IU	value

CDF Parameter of view	Value	Display
gain	5	off
gainv		off

Ensure that *CDF* is selected in the *Show* area and then examine the *CDF Parameter of view* cyclic field. By default, the *CDF Parameter of view* field is set to the name of the cellview bound with the instance you selected.

3. Change the parameter values as necessary.

Be aware that if a parameter has the same name in multiple cellviews, changing the value of the parameter in one cellview changes the value in all the cellviews that use the parameter.

4. Click *OK*.

### Changing the Parameters of a Cellview Not Currently Bound with an Instance

You can change the values of parameters in cellviews that are available for an instance but are not currently bound with the instance. Parameters changed in this way become effective only if you bind the changed cellview with the instance from which the cellview was changed. Associating the changed cellview with a different instance has no effect because cellview parameters are instance specific.

To change the values of parameters in cellviews that are available for an instance but not currently bound with the instance, follow these steps:

1. Select the instance in the Virtuoso Schematic Editing window.
2. Choose *Edit – Properties – Objects*.

The Edit Object Properties form opens.

3. Ensure that *CDF* is selected in the *Show* area and then set the *CDF Parameter of view* cyclic field to the cellview whose parameters you want to change.
4. Change the parameter values of the cellview as necessary.

Be aware that if a parameter has the same name in multiple cellviews, changing the value of the parameter in one cellview changes the value in all the cellviews that use the parameter.

5. Click *OK*.

### **Deleting Parameters from a veriloga Cellview**

To delete a parameter from a cellview, you must edit the original veriloga text module. Follow these steps:

1. In the Virtuoso Schematic Editing window, select an instance for which the Verilog-A cellview is available.
2. Choose *Design – Hierarchy – Descend Edit*.

The Descend form opens.

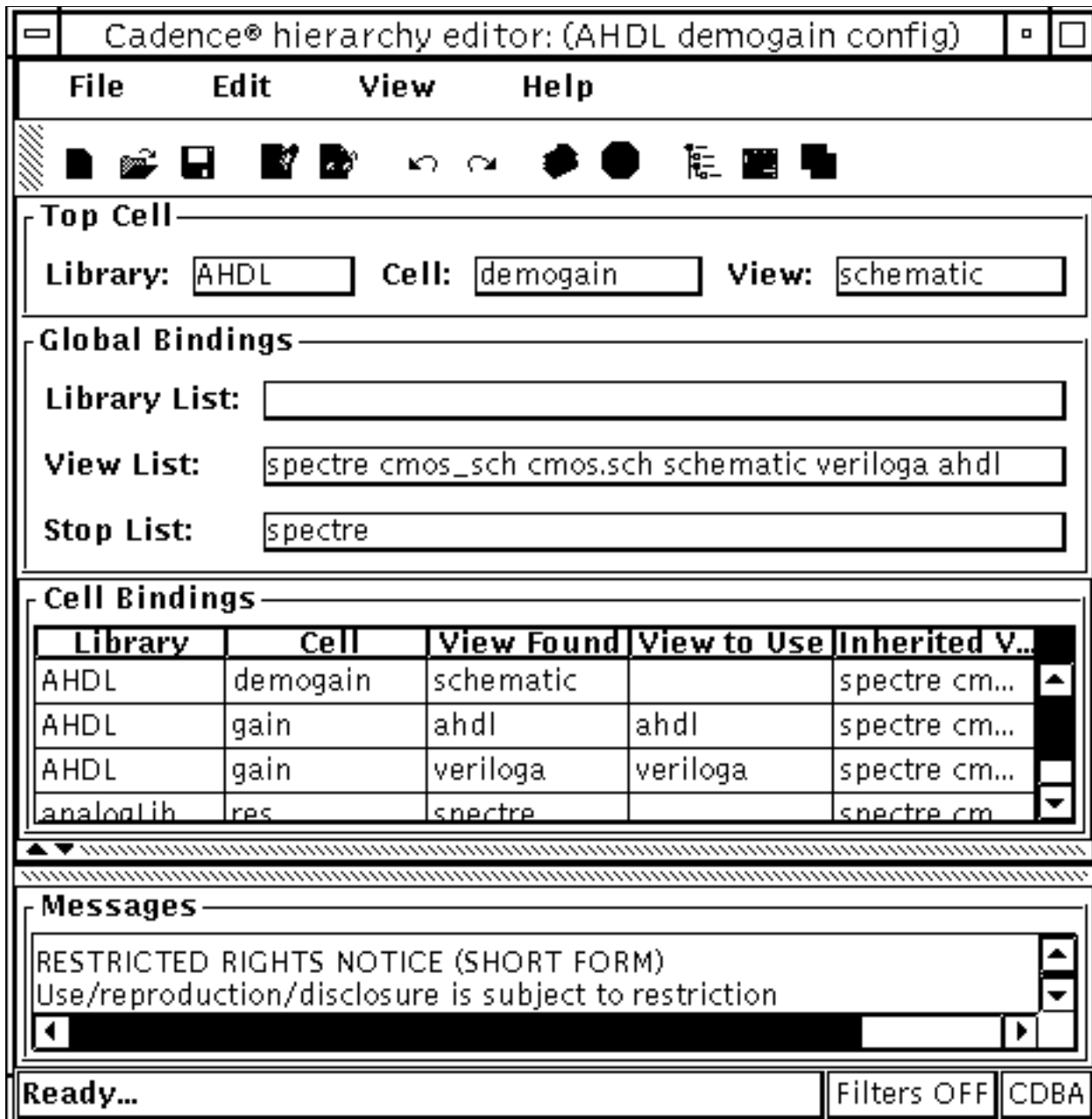
3. In the *View Name* cyclic field, choose the Verilog-A cellview that defines the parameter you want to delete.
4. Click *OK*.

A text editing window opens with the module text displayed.

5. Delete the parameter definition statement for the parameter you want to delete.
6. Save your changes and quit the text editing window.

## Switching the Cellview Bound with an Instance

There are several ways to bind different cellviews with particular instances. One way, described here, is to use the Cadence hierarchy editor window.



To specify the cellview that you want to bind with an instance, follow these steps:



## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

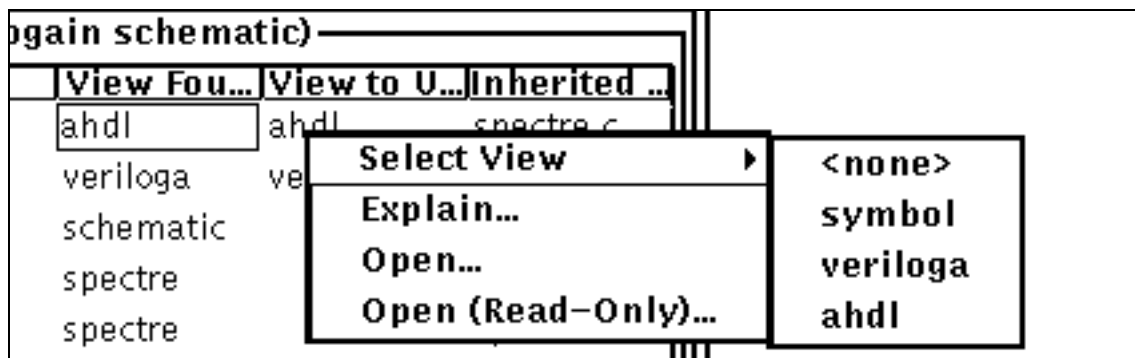
---

1. In the Cadence Hierarchy Editor window, choose *View* from the menu and turn on *Instance Table*.
2. In the *Cell Bindings* section, click the cell that instantiates the instance you want to switch.

The instances appear in the *Instance Bindings* section of the Cadence Hierarchy Editor window. The *View Found* column shows the cellview bound with each instance (the view that is selected for inclusion in the hierarchy).

3. Right click the *View To Use* table cell for the instance you want to switch.

A pop-up menu opens.



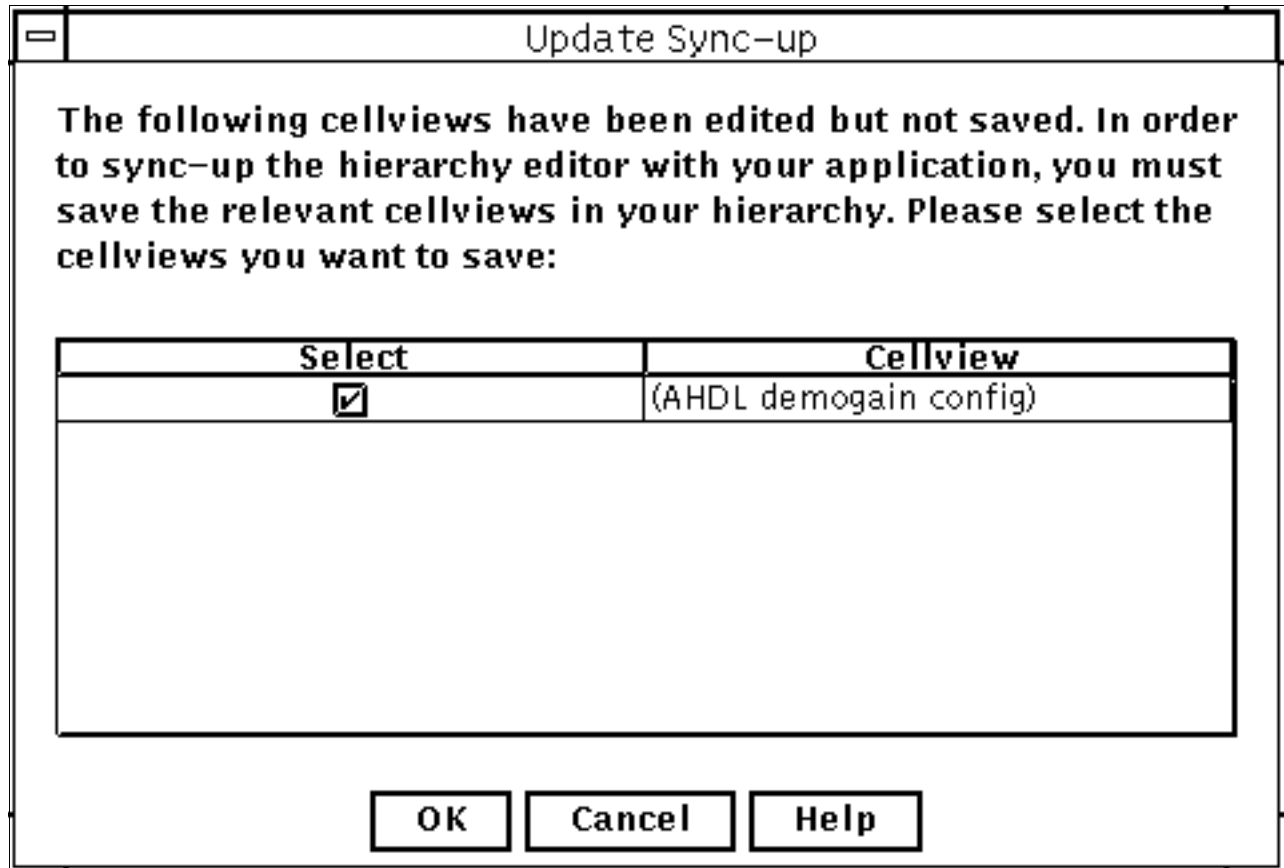
4. In the pop-up menu, choose *Select View* and the name of the cellview that you want to bind with the instance.

### Synchronizing the Schematic with Changes in the Hierarchy Editor

Whenever you switch cellviews in the Cadence hierarchy editor, you must synchronize the associated schematic. If you do not synchronize your schematic to the changed Cadence hierarchy editor information, your design does not netlist correctly. To ensure that the Cadence hierarchy editor and the Virtuoso Schematic Editing windows are synchronized, follow these steps:

1. In the Cadence hierarchy editor window, click the *Update (Needed)* button or choose *View – Update (Needed)* from the menu.

The Update Sync-up form opens.



2. Turn on the checkmarks by all the listed cellviews.
3. Click *OK*.

### Synchronizing the Hierarchy Editor with Changes in the Schematic

If you use the Virtuoso Schematic Editing window to add or delete an instance, you must synchronize the Cadence hierarchy editor by following these steps:

1. In the Virtuoso Schematic Editing window, choose *Design – Check and Save*.
2. If the *Hierarchy-Editor* menu entry is not visible, choose *Tools – Hierarchy Editor* to make the entry appear.
3. Choose *Hierarchy-Editor – Update*.

## Example Illustrating Cellview Switching

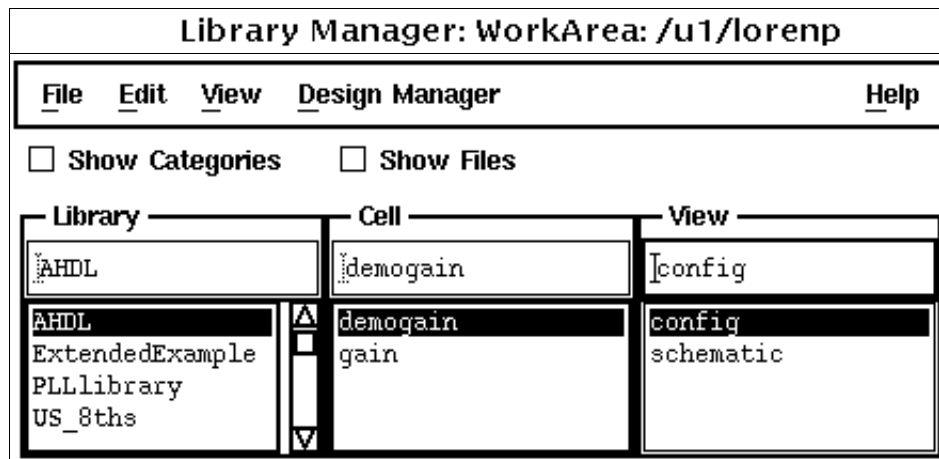
The following sections illustrate how cellview switching works. The example uses a circuit, called `demogain`, that consists of two instances of a module called `gain`, two resistors, and a power source. The two instances amplify the signal, with the output from the first instance becoming the input for the second. The `demogain` cell has both schematic and config views.

This example is not included in any supplied library. To use cellview switching in your own designs, follow steps similar to these, substituting your own modules and components.

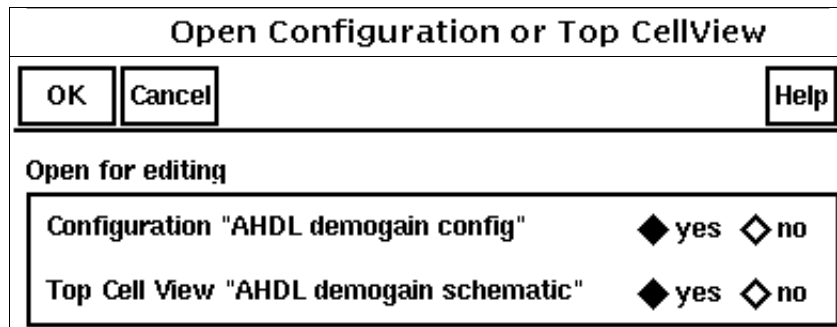
### Opening the Design

To open the schematic and config views for the `demogain` module, follow these steps:

1. In the CIW, choose *Tools – Library Manager*.
2. In the Library Manager window, select the `demogain` cell and the `config` view.



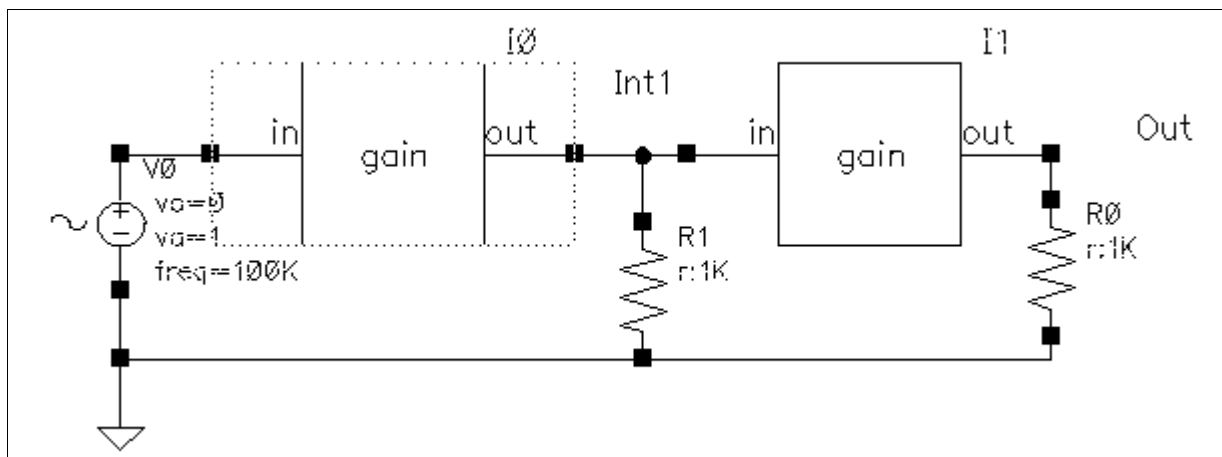
3. Choose *File – Open* and, when asked, indicate that you want to open both the config and schematic views.



### Examining the Text Module Bound with Instance I0

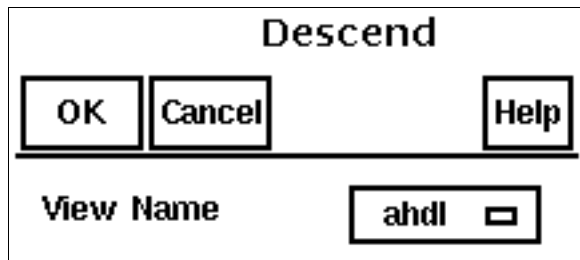
To examine the text module bound to instance I0, follow these steps:

1. In the Virtuoso Schematic Editing window, select I0, the first instance of the *gain* module.



2. From the menu bar, choose *Design – Hierarchy – Descend Edit*.

The Descend dialog box opens, with the *View Name* cyclic field showing the cellview currently bound with the selected instance.



3. Click *OK*.

The text module bound with `I0` appears. The module has two parameters: `gain`, with a value of 3, and `gainh`, with a value of 2.

4. Quit the text module window.

### Checking the Edit Object Properties Form for Instance I0

To examine the parameters currently in effect for instance `I0`, follow these steps:

1. With instance `I0` still selected, click *Property*.

The Edit Object Properties form opens.

**Edit Object Properties**

OK
Cancel
Apply
Defaults
Previous
Next
Help

**Apply To**    only current 
instance

**Show**         system    user    CDF

Browse
Reset Instance Labels Display

Property	Value	Display
Library Name	AHDL	off <input type="checkbox"/>
Cell Name	gain	off <input type="checkbox"/>
View Name	symbol	off <input type="checkbox"/>
Instance Name	IO	value <input type="checkbox"/>

CDF Parameter of view	ahdl <input type="checkbox"/>	Display
gain		off <input type="checkbox"/>
gainh		off <input type="checkbox"/>

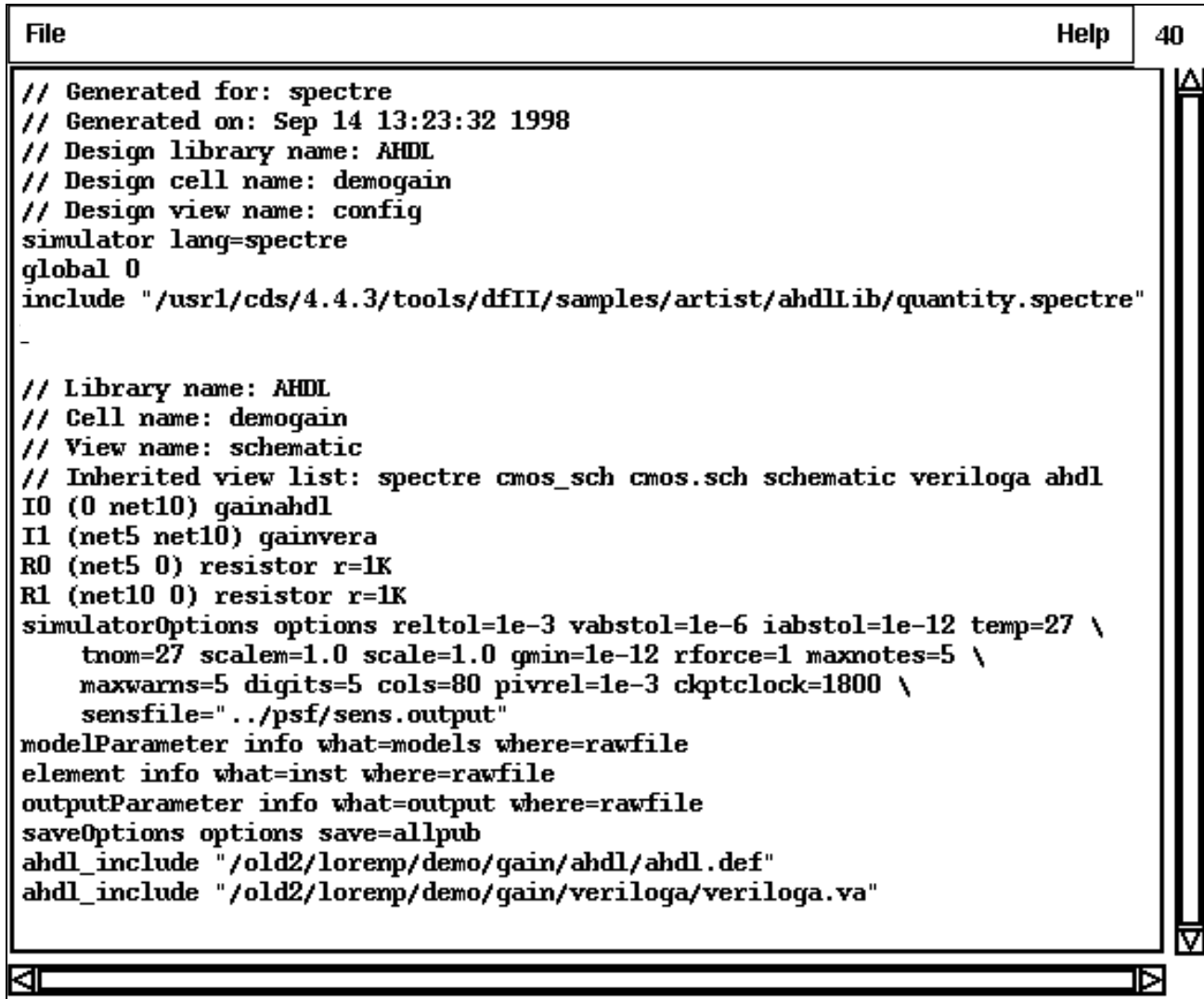
2. Ensure that *CDF* is selected in the *Show* area. The *gain* and *gainh* parameters are displayed without values because the values defined in the text modules are in effect.

## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

As a check, you can use the capabilities of the analog design environment Simulation window to generate a netlist.



```
File Help 40
// Generated for: spectre
// Generated on: Sep 14 13:23:32 1998
// Design library name: AHDL
// Design cell name: demogain
// Design view name: config
simulator lang=spectre
global 0
include "/usr1/cds/4.4.3/tools/dfII/samples/artist/ahdLib/quantity.spectre"
-
// Library name: AHDL
// Cell name: demogain
// View name: schematic
// Inherited view list: spectre cmos_sch cmos.sch schematic veriloga ahdl
I0 (0 net10) gainahdl
I1 (net5 net10) gainvera
R0 (net5 0) resistor r=1K
R1 (net10 0) resistor r=1K
simulatorOptions options reltol=1e-3 vabstol=1e-6 iabstol=1e-12 temp=27 \
    tnom=27 scalem=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 \
    maxwarns=5 digits=5 cols=80 pivrel=1e-3 clkclock=1800 \
    sensfile=" ../psf/sens.output"
modelParameter info what=models where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
saveOptions options save=allpub
ahdl_include "/old2/lorenp/demo/gain/ahdl/ahdl.def"
ahdl_include "/old2/lorenp/demo/gain/veriloga/veriloga.va"
```

The netlist shows that instance I1 is bound with the Verilog-A module, gainvera.

### Checking the Text Module and Edit Object Properties Form for Instance I1

If you examine the Verilog-A module bound with I1, following the same steps used for instance I0, you find that it has two parameters: gain and gainv.

```
/old2/lorenp/demo/gain/veriloga/veriloga.va
//VerilogA for AHDL, gain, veriloga
`include "constants.vams"
`include "disciplines.vams"
```

## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

```

module gainvera(out, in);
output out;
electrical out;
input in;
electrical in;
parameter real gainv = 4.0 ;
parameter real gain = 1.0 ;
analog
    V(out) <+ (gain*gainv)*V(in);
endmodule

```

Checking the Edit Object Properties form for instance I1 shows the *CDF Parameter of view* cyclic field set to *veriloga*, matching the Verilog-A code of the bound module. Again, no parameter values are displayed because the values defined in the text module are used.

Edit Object Properties

OK
Cancel
Apply
Defaults
Previous
Next
Help

Apply To     only current     instance

Show         system     user     CDF

Browse
Reset Instance Labels Display

Property	Value	Display
Library Name	AHDL	off <input type="checkbox"/>
Cell Name	gain	off <input type="checkbox"/>
View Name	symbol	off <input type="checkbox"/>
Instance Name	I1	value <input type="checkbox"/>

CDF Parameter of view	veriloga <input type="checkbox"/>	Display
gain		off <input type="checkbox"/>
gainv		off <input type="checkbox"/>



## Modifying Instance Parameters

Verilog-A modules contain default values for their parameters. These default values are used during netlisting unless you override them on the Edit Object Properties form or on the Edit Component CDF form. To change the two parameters used in the cellview bound with instance I0, follow these steps:

1. In the Virtuoso Schematic Editing window, select instance I0 and click *Property*.

The Edit Object Properties form opens.

**Edit Object Properties**

---

**Apply To**   

**Show**         system    user    CDF

---

Property	Value	Display
Library Name	AHDL	off <input type="checkbox"/>
Cell Name	gain	off <input type="checkbox"/>
View Name	symbol	off <input type="checkbox"/>
Instance Name	I0	value <input type="checkbox"/>

---

CDF Parameter of view	Value	Display
gain	5	off <input type="checkbox"/>
gainh	6	off <input type="checkbox"/>

2. Ensure that *CDF* is selected in the *Show* area.
3. Type 5 in the *gain* field and 6 in the *gainh* parameter field.
4. Click *OK* or *Apply*.

## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

If you generate a final netlist, you see that the value of `gain` in the netlist is now 5 and the value of `gainh` is now 6, as expected.



The screenshot shows a text editor window titled `/old2/lorenp/simulation/demogain/spectre/config/netlist/input.scs`. The window has a menu bar with `File` and `Help`, and a page number `41`. The main text area contains the following Verilog-A code:

```
// Generated for: spectre
// Generated on: Sep 14 13:29:12 1998
// Design library name: AHDL
// Design cell name: demogain
// Design view name: config
simulator lang=spectre
global 0
include "/usr1/cds/4.4.3/tools/dfII/samples/artist/ahdlLib/quantity.spectre"

// Library name: AHDL
// Cell name: demogain
// View name: schematic
// Inherited view list: spectre cmos_sch cmos.sch schematic veriloga ahdl
I0 (0 net10) gainahdl gain=5 gainh=6
I1 (net5 net10) gainvera
R0 (net5 0) resistor r=1K
R1 (net10 0) resistor r=1K
simulatorOptions options reltol=1e-3 vabstol=1e-6 iabstol=1e-12 temp=27 \
  tnom=27 scalem=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 \
  maxwarns=5 digits=5 cols=80 pivrel=1e-3 ckptclock=1800 \
  sensfile=" ../psf/sens.output"
modelParameter info what=models where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
saveOptions options save=allpub
ahdl_include "/old2/lorenp/demo/gain/ahdl/ahdl.def"
ahdl_include "/old2/lorenp/demo/gain/veriloga/veriloga.va"
```

### Associating New Cellviews with Instances I0 and I1

To switch the cellviews bound with instances `I0` and `I1`, follow these steps:

1. In the Cadence hierarchy editor window, click the *Instance Table* button to display the *Instance Bindings* table.
2. In the *Cell Bindings* table, click the cell containing `demogain`.

**Cadence Verilog-A Language Reference**  
Using Verilog-A in the Cadence Analog Design Environment

The instances within `demogain` appear in the *Instance Bindings* table.

Instance Bindings					
Library	AHDL	Cell	demogain	View	schematic
Inst Name	Library	Cell	View Found	View To Use	
I0	AHDL	gain	ahdl	ahdl	
I1	AHDL	gain	veriloga		
R0	analogLib	res	spectre		
R1	analogLib	res	spectre		
I2	basic	gnd	schematic		

3. In the *Instance Bindings* table, right click on the *View To Use* entry for the I0 instance of cell `gain`.
4. From the pop-up menu, choose *Select View – veriloga*.  
The *View Found* and the *View To Use* fields both change to `veriloga`.
5. In the *Instance Bindings* table, right click on the *View To Use* entry for the I1 instance of cell `gain`.
6. From the pop-up menu, choose *Select View – ahdl*.  
The *View Found* and the *View To Use* fields both change to `ahdl`.
7. In the Cadence hierarchy editor window, click the *Update (Needed)* button.  
The Update Sync-up form appears.
8. Turn on the checkmarks next to the changed cells.
9. Click *OK*.

### Parameter Values after Switching the Cellview Bound with Instance I0

As noted in “[Changing the Parameters of a Cellview Not Currently Bound with an Instance](#)” on page 270, cellview parameters are instance specific. To demonstrate this with the example, follow these steps:

1. In the Virtuoso Schematic Editing window, select instance `I0` and click *Property*.

The Edit Object Properties form opens.

2. Ensure that *CDF* is selected in the *Show* area, and look at the *CDF Parameter of view* cyclic field.

The cyclic field shows *veriloga* because the veriloga cellview is currently bound with instance `I0`. Recall that when the parameter values were set for instance `I0`, the bound cellview was *ahdl*, not *veriloga*.

3. Switch the *CDF Parameter of view* field to *ahdl*.

The parameter values set for instance `I0` while it was bound with the *ahdl* cellview appear. If you rebind the *ahdl* cellview with instance `I0`, the *ahdl* parameter values take effect again.

4. Switch the *CDF Parameter of view* field back to *veriloga*.

The `gain` parameter has a value of 5. It has this value because the `gain` parameter occurs in both the *veriloga* and *ahdl* cellviews. When `gain` in the *ahdl* cellview was given a value, the `gain` parameter in the *veriloga* cellview took on the same value. If you change a shared parameter such as `gain` in one cellview, the value changes in other cellviews of the same component that share the parameter.

Generating another final netlist for this switched cellview design confirms that the `I0` instance is bound with the veriloga cellview. The netlist also shows that the `gain` parameter has the expected value of 5.

```
// Generated for: spectre
// Generated on: Sep 14 10:27:48 1998
// Design library name: AHDL
// Design cell name: demogain
// Design view name: config
simulator lang=spectre
global 0
include "/usr1/cds/4.4.3/tools/dfII/samples/artist/ahdllib/quantity.spectre"

// Library name: AHDL
// Cell name: demogain
// View name: schematic
// Inherited view list: spectre cmos_sch cmos.sch schematic veriloga ahdl
I0 (net10 0) gainvera gain=5
I1 (net10 net5) gainahdl
R0 (net5 0) resistor r=1K
R1 (net10 0) resistor r=1K
simulatorOptions options reltol=1e-3 vabstol=1e-6 iabstol=1e-12 temp=27 \
    tnom=27 scalem=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 \
    maxwarns=5 digits=5 cols=80 pivrel=1e-3 ckptclock=1800 \
    sensfile=" ../psf/sens.output"
modelParameter info what=models where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
saveOptions options save=allpub
ahdl_include "/old2/lorenp/demo/gain/ahdl/ahdl.def"
ahdl_include "/old2/lorenp/demo/gain/veriloga/veriloga.va"
```

## Multilevel Hierarchical Designs

You can use Verilog-A modules inside a multilevel design hierarchy in the following ways:

- Instantiate child Verilog-A modules inside parent analog HDL modules
- Place a Verilog-A cellview instance in a schematic design
- Instantiate a schematic in a Verilog-A module

You can use any number of levels of hierarchy with schematic and Verilog-A cellviews at any level, but you cannot pass parameters down to levels that are lower than the first point where a component with a schematic cellview occurs below a component with a Verilog-A cellview.

When a design with Verilog-A cellviews is netlisted, no additional action is required. Verilog-A modules can also be included through the Model Library Setup form. This is described in the next section.

## Including Verilog-A through Model Setup

In some situations, you might need to explicitly include Verilog-A modules. For example, you want a module definition for a device referenced through the model instance parameter. In this case, you must specify a file through the Model Library Setup form, which includes the files with the Verilog-A definitions.

## Netlisting Verilog-A Modules

Verilog-A modules are included in netlists through the use of a special `include` statement. The statement has this format:

```
ahdl_include "filename"
```

For example, if you have an analogLib npn instance with the *Model Name* set to `ahdlNpn`, the file `includeHDLs.scs` has the line `ahdl_include "/usr/ahdlNpn.va"`. The file `includeHDL.scs` is entered on the Model Library Setup form.

Use full UNIX paths that resolve across your network for filenames. For more information about specifying filenames, see the *Cadence Analog Design Environment User Guide*. For a Verilog-A file, *filename* must have a `.va` file extension.

## Hierarchical Verilog-A Modules

You can create a hierarchy in a Verilog-A module by instantiating lower-level modules inside a higher-level module. You can instantiate Spectre primitives, Verilog-A modules, and schematics inside a Verilog-A module. The netlister automatically adds the necessary `ahdl_include` statements in the netlist for each Verilog-A module, including modules within a module. For example, in the following module, one module, `VCOshape`, is instantiated inside (below) another, `VCO2`.

```
module VCO2(R1, ref, out, CA, CB, VCC, vControl);
inout R1, ref, out, CA, CB, VCC, vControl;
electrical R1, ref, out, CA, CB, VCC, vControl;
electrical cntrl;
real state;
VCOshape shape (ref, cntrl, VCC, vControl);
resistor #(.r(0.001)) RX(CB, ref);
```

## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

```
resistor #(.r(500)) R1min(cntrl, R1);
capacitor #(.c(10p)) Cmin (CA, CB);
analog begin
@(initial_step) begin
    state = 1.0;
end
if (analysis("dc") || $abstime == 0.0)
    val(CA, CB) <+ 0.0;
if (@(cross(val(CA)+1.0, -1)))
    state = 1.0;
if (@(cross((val(CA)-1.0, +1)))
    state = -1.0;
I(CA) <+ -(1.71*I(cntrl, R1)*val(VCC, ref)*val(out));
val(out) <+ $transition(state, 10n, 10n, 10n);
end
endmodule
```

## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

The VCO2 module is part of a larger schematic, which produces the following netlist:

Instantiation of VCO2 in the top-level design

```
// Generated for: spectre
// Generated on: Aug 20 07:32:00 1998
// Design library name: QPSK
// Design cell name: Example24_VCOQuad
// Design view name: schematic
simulator lang=spectre
global 0

// Library name: QPSK
// Cell name: Example24_VCOQuad
// View name: schematic
VCTRL (vc 0) vsource type=sine sinedc=3 ampl=2 freq=500K
C12 (ca cb) capacitor c=20p
I11 (r1 0 out ca cb VCC vc) VCO2
I9 (outi outq out) quadrature riseTime=10n
R7 (r1 0) resistor r=2.2K
vcc (VCC 0) vsource dc=6 type=dc
simulatorOptions options reltol=1e-3 vabstol=1e-6 iabstol=1e-12 temp=27 \
    tnom=27 scalem=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 maxwarns=5 \
    digits=5 cols=80 pivrel=1e-3 ckptclock=1800 \
    sensfile="..psf/sens.output"
dcOp dc write="spectre.dc" oppoint=rawfile maxiters=150 maxsteps=10000 \
    annotate=status
modelParameter info what=models where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
saveOptions options save=allpub
ahdl_include "/net/cds9886/u1/public/ahdl/demo/QPSK/VCO2/ahdl/ahdl.def"
ahdl_include "/net/cds9886/u1/public/ahdl/demo/QPSK/VCOshape/ahdl/ahdl.def"
ahdl_include "/net/cds9886/u1/public/ahdl/demo/QPSK/quadrature/ahdl/ahdl.def"
```

The netlister automatically creates `ahdl_include` statements for VCO2 and VCOshape.

## Using a Hierarchy

You can add symbols that have a Verilog-A cellview to any schematic, but you cannot add a child Verilog-A module to a schematic without a corresponding symbol view. To ensure proper binding, you must create the symbol view before you create the Verilog-A module or, once you have created both the Verilog-A view and the symbol view, reopen the Verilog-A view and write it again. If the design is structured in multiple levels, you can include components with



# Cadence Verilog-A Language Reference

## Using Verilog-A in the Cadence Analog Design Environment

---

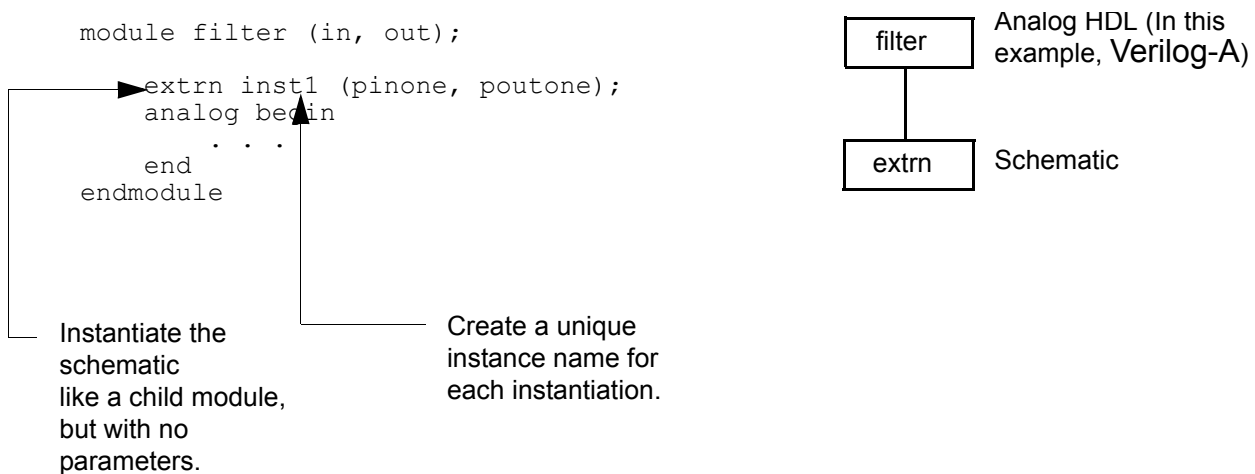
Verilog-A views below a schematic level, and you can include components with schematic views below Verilog-A components.

You can instantiate schematics in Verilog-A modules, but there are two important rules you must remember:

- The Spectre simulator cannot pass parameters to a schematic that is a child module (a module within another module).
- When instantiating a schematic inside a module, the cell that the schematic represents must also have a symbol view for the design to netlist correctly.

If you do not use a schematic from the same library as the Verilog-A module, the analog design environment searches every library and uses the first cell it finds that has the same name.

A schematic placed below a Verilog-A module can include other schematics or Verilog-A views.



### Simulation View Lists

If you examine the Environment Options form, by choosing *Setup – Environment* in the simulation control window, you see `veriloga` and `ahdl` in *Switch View List*. By default, `ahdl` is in the last position and `veriloga` is assigned the next to last position.

*Switch View List*

```
spectre cnos_sch cnos.sch schematic veriloga ahdl
```

*Stop View List*

```
spectre
```

# Cadence Verilog-A Language Reference

## Using Verilog-A in the Cadence Analog Design Environment

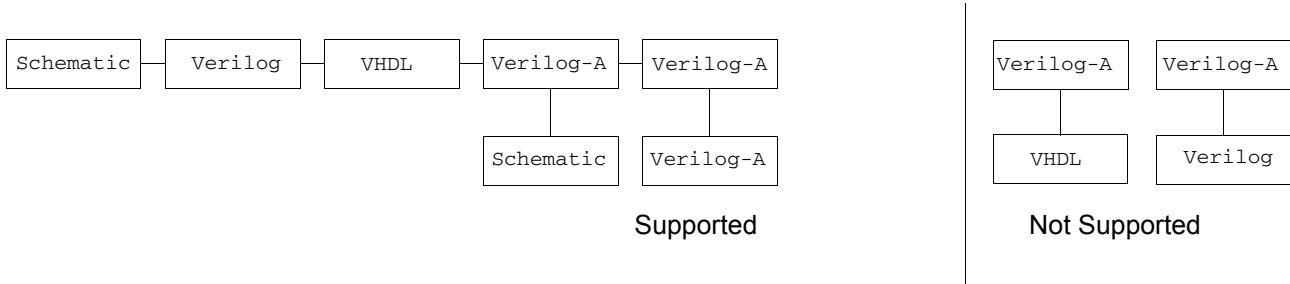
---

If you create cellviews with names other than the default names (for example, `veriloga_2`), you must adjust the view lists to netlist properly.

In mixed-signal mode, or to create analog configurations, use the Cadence hierarchy editor to modify *Switch View List* and *Stop View List*.

### Verilog and VHDL

The same component can have digital Verilog and VHDL cellviews as well as Verilog-A cellviews. You can wire symbols with Verilog or VHDL cellviews to symbols with Verilog-A cellviews in the same schematic. You cannot instantiate a Verilog or VHDL file inside or below an Verilog-A module.



## Using Models with Verilog-A

Verilog-A supports the use of models inside of modules. In a Verilog-A module, you can instantiate any Spectre primitive based on a model.

### Models in Modules

When using models in a Verilog-A module, you treat the models as child modules. You instantiate each instance of the model in a single statement with the model name, the instance name, the node list, and the parameter list.

Two instances of the same model, with parameter passing

```
module dual_npn (c1, c2, b1, b2, e, s) ;
  electrical c1, c2, b1, b2, e, s ;
  parameter real a = 1 ;
  my_npn #(.a(1.0)) q0 (c1, b1, e, s) ;
  my_npn #(.a(1.0)) q1 (c2, b2, e, s) ;
endmodule
```

The models are included through one of the files specified in the Model Library Setup form.

**Note:** For spectreS, for each model you use, you must have a corresponding model file. To reference that file, you must specify the model file as an include file by choosing *Setup – Simulation Files – Include File* in the Cadence analog design environment Simulation window.

**Note:** For spectreS, the model file must have a `.m` file extension. The contents of the model file follow SPICE syntax unless you switch the language inside of the model file to Spectre syntax.

## Saving Verilog-A Variables

When you want to plot or display the values of internal Verilog-A variables, you can specify which variable to save as shown in [step 4](#) in the following section. To plot or display all Verilog-A variables, you can save them all with one simple option:

```
Saveahdl options saveahdlvars=all
```

In this case, no explicit save needs to be done.

To save all module parameters in the Cadence analog design environment using the spectre/spectreVerilog interface, do the following:

- In the simulation control window, choose *Outputs – Save All*. The *Outputs – Save All* command opens the Save Options form. On that form, click *all* (located next to *Select AHDL variables (saveahdlvars)*).

## Displaying the Waveforms of Variables

To plot the value of a Verilog-A variable, follow these steps:

1. Find the instance names of each Verilog-A module that contains variables that you want to plot.
2. In the Cadence analog design environment Simulation window, choose *Setup – Model Libraries*.

The Setup – Model Libraries form opens.

3. Enter the full UNIX path of the file. For more information about specifying filenames, see the *Cadence Analog Design Environment User Guide*.
4. Edit the file. Type

```
save instance_name:variable_name
```

## Cadence Verilog-A Language Reference

### Using Verilog-A in the Cadence Analog Design Environment

---

*instance\_name* is the full hierarchical name described in step 1 or 2.  
*variable\_name* can be `all`, if you want to prepare to display all variables, or a specific variable name.

Use the following syntax for the hierarchical name of the instance:

```
hier_name ::=  
    [ instance_name{.instance_name}.]HDL_Instance_name
```

Provide *instance\_name* only if the Verilog-A instance is embedded within a hierarchical design.

You find *instance\_name* and *HDL\_Instance\_name* in the schematic editor's Edit Object Properties form. *instance\_name* is the value in the *Instance Name* field. See the following examples of hierarchical instances.

Verilog-A instance below two blocks	—————▶	<code>i7.i2.i3</code>
Verilog-A instance below one block	—————▶	<code>i2.i3</code>

In the previous examples, `i7` and `i2` represent instances of schematic cellviews, and `i3` represents an instance of a Verilog-A cellview.

**Note:** The syntax for internal nodes is

```
save instance_name.internal_node_name
```

See the [Spectre Circuit Simulator User Guide](#) for more information about the `save` statement.

5. Run the simulation.
6. In the simulation control window, choose *Tools – Results Browser*.

The system prompts you for a project directory.

7. Type

```
simulation/design_name/spectre/view_name
```

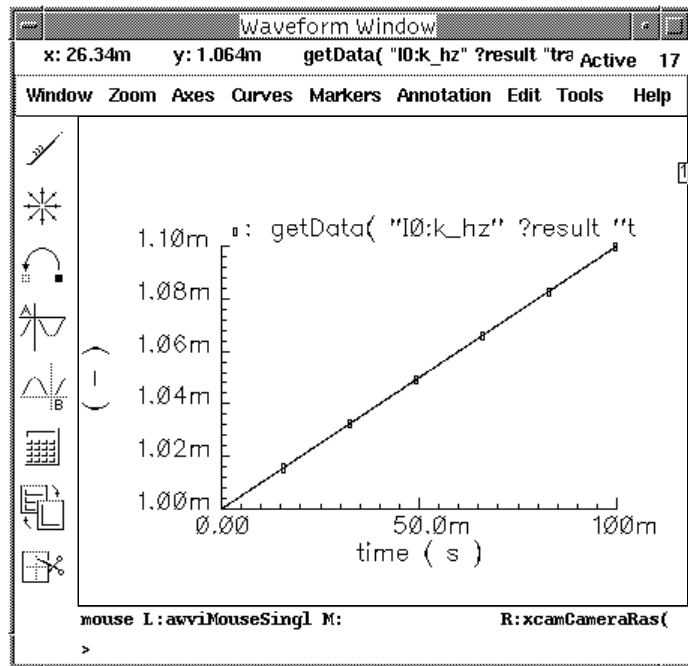
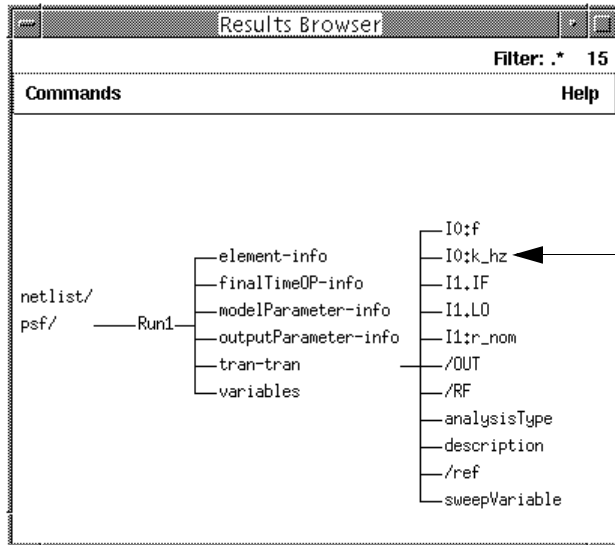
where *design\_name* is the name of your design and *view\_name* is the name of your cellview.

8. Open the *psf* portion of the output database and search for the variable name you identified for the analysis you ran.

# Cadence Verilog-A Language Reference

## Using Verilog-A in the Cadence Analog Design Environment

9. When you find the variable name in the Browser, use the menu option *Plot* (on the middle mouse button) to plot the output from the variable.



**Cadence Verilog-A Language Reference**  
Using Verilog-A in the Cadence Analog Design Environment

---

---

## Verilog-A Modeling Examples

---

You can use the Cadence® Verilog®-A language to model complex systems. See the following topics for some examples:

- [Electrical Modeling](#) on page 296
  - [Three-Phase, Half-Wave Rectifier](#) on page 296
  - [Thin-Film Transistor Model](#) on page 301
- [Mechanical Modeling](#) on page 307
  - [Car on a Bumpy Road](#) on page 308
  - [Gearbox](#) on page 315

See also [“Computing a Moving or Sliding-Window Average”](#) on page 321.

## Electrical Modeling

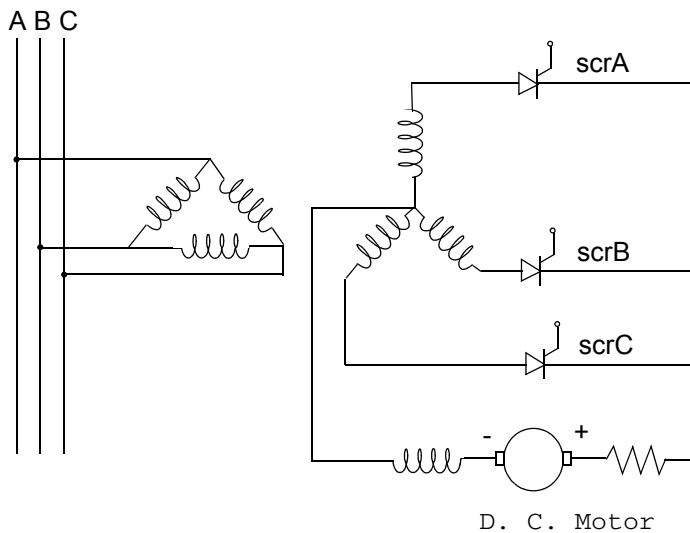
This section presents examples that illustrate the power and flexibility of Verilog-A when used to model electrical systems. The examples illustrate the analysis and behavioral modeling capabilities of Verilog-A.

- The first example shows how to use Verilog-A to model a rectifier. This example demonstrates how to use Verilog-A in the design of power circuits.
- The second example shows how to create a detailed model of a thin-film transistor using Verilog-A.

### Three-Phase, Half-Wave Rectifier

The following circuit converts the three-phase, AC line voltages into a rectified signal that produces a DC current to drive a motor. The speed of the motor is linearly related to the amplitude of this current. You can control the amplitude of the current by delaying the thyristor switching.

#### Rectifier Circuit



#### Operation

To understand the operation of this circuit, consider how the circuit functions if the thyristors are replaced by diodes. All three diodes have the same cathode node. The diodes are nonlinear and their conductance increases with the voltage across them. The diode with the largest anode voltage conducts while the other two stay off.



## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

If the anode voltage of one of the nonconducting diodes rises above that of the conducting diode, the current diverts to the diode with the higher anode voltage. In this way, the voltage at the common cathode always equals the maximum of the diode anode voltages minus the diode voltage drop.

Assuming that the inductance of the load is large, the current flowing in the load remains constant while it switches between the different diodes.

The thyristor differs from the diode in having a third terminal. Unlike the diode, the thyristor does not conduct when its anode voltage exceeds its cathode voltage. To cause the device to conduct, a pulse is required at the gate input of the thyristor. The thyristor continues to conduct current even after this pulse has been removed, as long as the current flowing through it is greater than a hold value.

The gate terminal on the thyristor allows the current switching to be delayed with respect to the diode switching points. By delaying the gate pulses, you can vary both the average DC voltage at the output and the average load current.

### Modeling

The following Verilog-A module models the thyristor. The thyristor is modeled as a switch that closes when its gate is activated and opens when the current flowing through it falls below the hold value. When the thyristor is conducting, it has a nonlinear resistance. Without the nonlinearity, the circuit does not function correctly. The nonlinear resistance ensures that the thyristor with the largest anode voltage conducts all the current when its gate is activated.

```
module thyristor(anode, cathode, gate);
input gate;
inout anode, cathode;
electrical anode, cathode, gate;
parameter real vtrigger = 2.0 from [0:inf);
parameter real ihold = 10m from [0.0:inf);
parameter real Rscr = 10;
parameter real Von = 1.3;

    integer thyristorState;
    analog begin

        // get simulator to place a breakpoint when V(gate)
        // rises past vtrigger

        @ ( cross( V(gate) - vtrigger, +1 ) )
            ;

        // get simulator to place a breakpoint when
        // I(anode,cathode) falls below ihold

        @ ( cross( I(anode,cathode) - ihold, -1 ) )
            ;
    end
endmodule
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

```
// now see if thyristor is beginning to conduct, or
// is turning off

if ( V(gate) > vtrigger ) begin
    thyristorState = 1;
end else if ( I(anode,cathode) < ihold ) begin
    thyristorState = 0;
end

// drive output. if conducting, use a non-linear
// resistance. if not-conducting, then open completely
// (no current flow)

if ( thyristorState == 1 ) begin
    V(anode,cathode) <+ I(anode,cathode) *
        Rscr * exp(-V(anode,cathode) );
end else if ( thyristorState == 0 ) begin
    I(anode,cathode) <+ 0.0;
end
end
endmodule
```

The transformers are modeled with the following module, which includes leakage inductance effects:

```
module tformer(inp, inm, outp, outm);
input inp, inm ;
output outp ;
inout outm;
electrical inp, inm, outp, outm;
parameter real ratio = 1 from (0:inf);
parameter real leakL = 1e-3 from [0:inf);

    electrical node1;

    analog begin
        V(node1, outm) <+ leakL*ddt(I(node1, outm));
        V(outp, node1) <+ ratio*V(inp, inm);
    end

endmodule
```

The module `half_wave` describes the rectifier circuit, which consists of three transformers and three thyristors.

```
`define LK_IND 30m          // leakage inductance

module half_wave( common, out, gnd, inpA, inpB, inpC, gateA, gateB, gateC );
electrical common, out, gnd, inpA, inpB, inpC, gateA, gateB, gateC;
parameter real vtrigger = 0.0;
parameter real ihold = 1e-9;
parameter integer w1 = 1 from [1:inf);    // num of primary windings
parameter integer w2 = 1 from [1:inf);    // num of secondary windings

    electrical nodeA, nodeB, nodeC;

    thyristor #(.vtrigger(vtrigger),.ihold(ihold))
        scrA(nodeA, out, gateA);
    thyristor #(.vtrigger(vtrigger),.ihold(ihold))
        scrB(nodeB, out, gateB);
```

## Cadence Verilog-A Language Reference

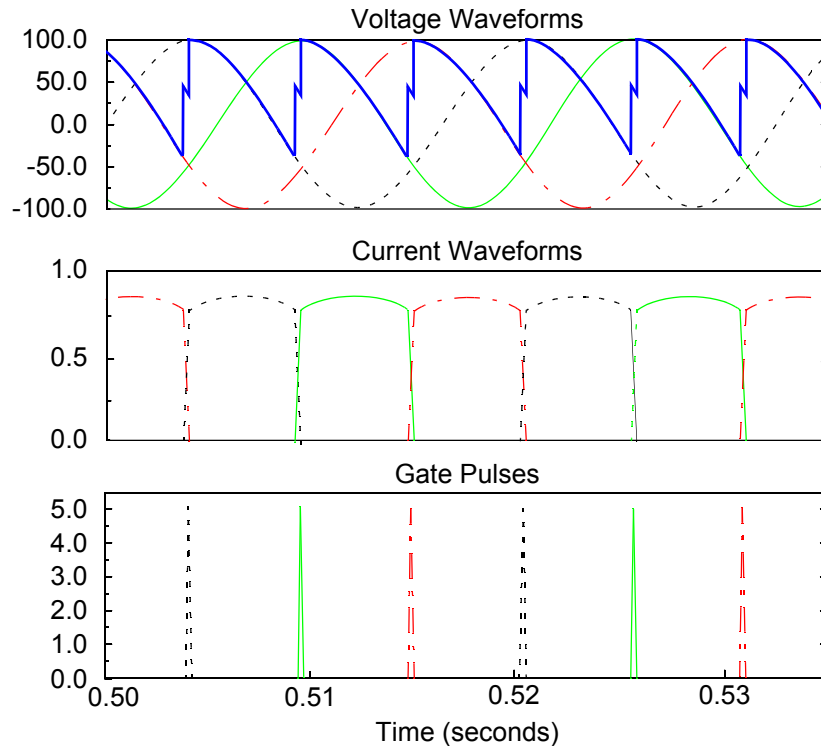
### Verilog-A Modeling Examples

```
thyristor #(.vtrigger(vtrigger),.ihold(ihold))
           scrC(nodeC, out, gateC);

tformer #(.ratio(w2/w1),.leakL(`LK_IND)) tA(inpA, gnd,
           nodeA, common);
tformer #(.ratio(w2/w1),.leakL(`LK_IND)) tB(inpB, gnd,
           nodeB, common);
tformer #(.ratio(w2/w1),.leakL(`LK_IND)) tC(inpC, gnd,
           nodeC, common);

endmodule
```

The first graph in the following figure shows the output voltage waveform (the thick, choppy line) superimposed on the three input voltage waveforms. The second graph displays the thyristor current waveforms and the third graph shows the gate pulses. The current switching occurs past the point where ordinary diodes would switch. This delayed switching reduces the average DC voltage across the load.



The output voltage stays at an average value for a short time during the switching. This corresponds to the overlap angle in the current waveforms caused by the transformer leakage inductance, which prevents the current in any thyristor from changing instantaneously. During the overlap angle, two thyristors are active, and their cathode voltage is the average of their anode voltages. Eventually, one of the thyristors switches off so that all the current flows through one device.

## Cadence Verilog-A Language Reference

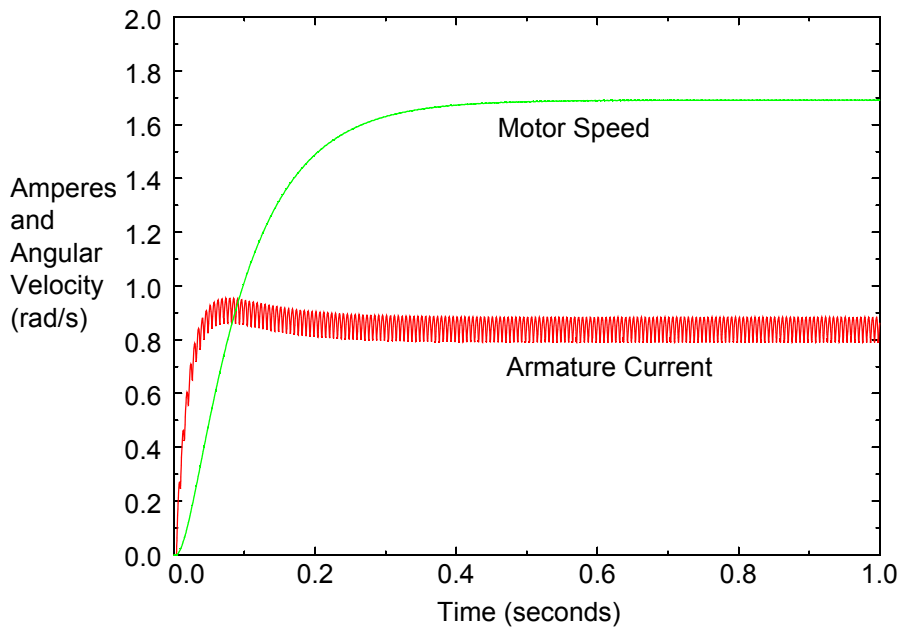
### Verilog-A Modeling Examples

---

The current remains almost constant, alternating through the three thyristors. During switching overlap, the current is shared between two thyristors. However, their sum remains almost constant.

The following figure shows the current to the load and the motor speed at startup. The module describing the motor is below the figure. Note how the module defines two internal nodes for speed and armature\_current, which can be plotted as node voltages.

**System Behavior at Startup Time**



```
module motor(vp, vn, shaft);
inout vp, vn, shaft;
electrical vp, vn;
rotational_omega shaft;
parameter real Km = 4.5 ; // motor constant [Vs/rad]
parameter real Kf = 6.2 ; // flux constant [Nm/A]
parameter real j = 0.004 ; // inertia factor [Nms2/rad]
parameter real D = 0.1 ; // drag (friction) [NMs/rad]
parameter real Rm = 5.0 ; // motor resistance [Ohms]
parameter real Lm = 1 ; // motor inductance [H]

electrical speed;
electrical armature_current;

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

endmodule
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

The Verilog-A modules described are assumed to be in a file called `rectifier_and_motor.va`, which includes the `disciplines.vams` file and the modules listed above in the same order as presented. The following Spectre netlist instantiates all the modules in this design. The motor shaft is left as an open circuit and simulated with no load. All the motor torque goes to overcome the inertia and windage losses. The `errpreset=conservative` statement in the `tran` line directs the simulator to use a conservative set of parameters as convergence criteria.

```
// motor netlist //
global gnd
simulator lang=spectre
ahdl_include "rectifier_and_motor.va"

#define FREQ 60
#define PER 1.0/60
#define DT PER/20 + PER/6
#define VMAX 100
#define STOPTIME 1

vA (inpA gnd) vsource type=sine freq=FREQ ampl=VMAX sinephase=0
vB (inpB gnd) vsource type=sine freq=FREQ ampl=VMAX sinephase=120
vC (inpC gnd) vsource type=sine freq=FREQ ampl=VMAX sinephase=240

vgA (gateA gnd) vsource type=pulse period=PER \
width=1u val0=0 val1=5 delay=DT
vgB (gateB gnd) vsource type=pulse period=PER \
width=1u val0=0 val1=5 delay=DT +2*PER/3
vgC (gateC gnd) vsource type=pulse period=PER \
width=1u val0=0 val1=5 delay=DT +PER/3

rect (gnd out gnd inpA inpB inpC gateA gateB gateC) half_wave

amotor out gnd shaft motor Rm=50 Lm=1 j=0.05 D=0.5 Kf=1.0
saveNodes options save=all
tran tran stop=STOPTIME start=-PER/24 errpreset=conservative
```

## Thin-Film Transistor Model

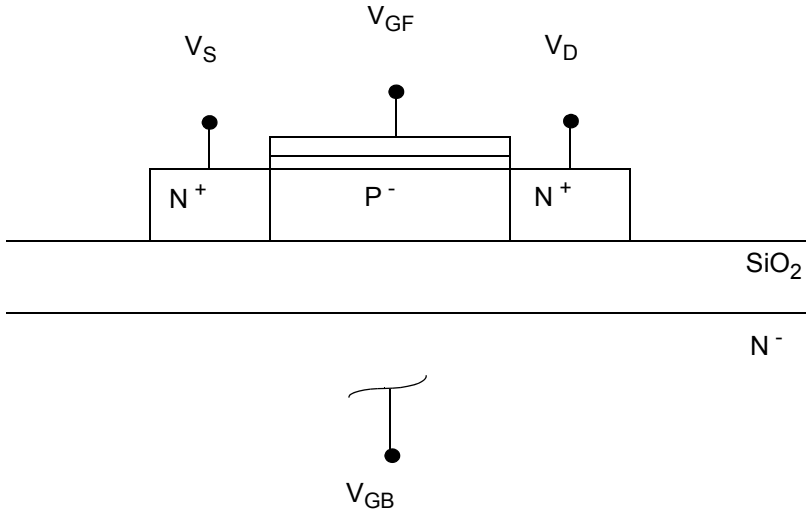
Verilog-A can support very detailed models of solid-state devices, such as a thin-film MOSFET, or TFT. The following figure shows the physical structure of a four-terminal, thin-film MOSFET transistor. The P-body region of the transistor is assumed to be fully depleted,

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

so both the front and back gate potentials influence channel conductivity. This implementation does not model short-channel effects.



The module definition is

```

`include "disciplines.vams"
`include "constants.vams"

`define CHECK_BACK_SURFACE 1
`define n_type 1
`define p_type 0

// "tft.va"
//
// mos_tft
//
// A fully depleted back surface tft MOSFET model. No
// short-channel effects.
//
// vdrain:      drain terminal      [V,A]
// vgate_front: front gate terminal [V,A]
// vsourCe:     source terminal     [V,A]
// vgate_back:  back gate terminal  [V,A]
//
//
module mos_tft(vdrain, vgate_front, vsource, vgate_back);
inout vdrain, vgate_front, vsource, vgate_back;
electrical vdrain, vgate_front, vsource, vgate_back;
parameter real length=1 from (0:inf);
parameter real width=1 from (0:inf);
parameter real toxf = 20n;
parameter real toxb = 0.5u;
parameter real nsub = 1e14;
parameter real ngate = 1e19;
parameter real nbody = 5e15;

```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

```
parameter real tb = 0.1u;
parameter real u0 = 700;
parameter real lambda = 0.05;
parameter integer dev_type=`n_type;

real
    id,
    vgfs,
    vds,
    vgbs,
    vdsat;

real
    phi, // body potential.
    vfbf, // flat-band voltage - front channel.
    vfbb, // flat-band voltage - back channel.
    vtfa, // threshold voltage - back channel accumulated.
    vgba, // vgb for accumulation at back surface.
    vgbi, // vgb for inversion at back surface.
    vtff, // threshold voltage.
    wkf, // work-function, front-channel.
    wkb, // work-function, back-channel.
    alpha, // capacitance ratio.
    cob, // capacitance back-gate to body.
    cof, // capacitance front-gate to body.
    cb, // body intrinsic capacitance.
    cbb, // series body / back-gate capacitance.
    cfb, // series front-gate / body capacitance.
    cfbb, // series front-gate / body / back-gate capacitance.
    qb, // fixed depleted body charge.
    kp, // K-prime.
    qgf, // front-gate charge.
    qgb, // back-gate charge.
    qn, // channel charge.
    qd, // drain component of channel charge.
    qs; // source component of channel charge.

integer back_surf;
real Vt, eps0, charge, boltz, ni, epsox, epsil;
real tmp1;
integer dev_type_sign;

analog begin

// perform initializations here

@ ( initial_step or initial_step("static") ) begin

    if( dev_type == `n_type ) dev_type_sign = 1;
    else dev_type_sign = -1;

    ni = 9.6e9; // 1/cm^3

    epsox = 3.9*`P_EPS0;
    epsil = 11.7*`P_EPS0;

    phi = 2*$vt*ln(nbody/ni);
    wkf = $vt*ln(ngate/ni) - phi/2;
    wkb = $vt*ln(nsub/ni) - phi/2;
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

```
vfbf = wkf;           // front-channel fixed charge assumed zero.
vfbb = wkb;           // back-channel fixed charge assumed zero.
qb = charge*nbody*1e6*tb;
cob = epsox/toxb;
cof = epsox/toxf;
cb = epsil/tb;
cbb = cob*cb/(cob + cb);
cfb = cof*cb/(cof + cb);
cfbb = cfb*cob/(cfb + cob);
alpha = cbb/cof;

vtfa = vbf + (1 + cb/cof)*phi - qb/(2*cof);
vgba = dev_type_sign*vfbb - phi*cb/cob - qb/(2*cob);
vgbi = dev_type_sign*vfbb + phi - qb/(2*cob);

kp = width*u0*1e-4*cof/length;

back_surf = 0;

end    // of initial_step code

// the following code is executed at every iteration

vgfs = dev_type_sign*V(vgate_front, vsource);
vds = dev_type_sign*V(vdrain, vsource);
vgbs = dev_type_sign*V(vgate_back, vsource);

// calc. threshold and saturation voltages.
//
vtff = vtfa - (vgbs - vgba)*cbb/cof;
vdsat = (vgfs - vtff)/(1 + alpha);

//
// drain current calculations.
//
if (vgfs < vtff) begin

    //
    // front-channel in accumulation / cutoff region(s).
    //
    id = 0;
    qn = 0;
    qd = 0;
    qs = 0;
    qgf = width*length*cfbb*(vgfs - wkf - qb/(2*cbb)
        - (vgbs - vfbb + qb/(2*cob)));
    qgb = - (qgf + width*length*qb);

end else if (vds < vdsat) begin

    //
    // front-channel in linear region.
    //
    id = kp*((vgfs - vtff)*vds - 0.5*
        (1 + cbb/cof)*vds*vds);
    id = id*(1 + lambda*vds);
    tmp1 = (1 + alpha)*vds;
    qn = -width*length*cof*(vgfs - vtff - tmp1/2 +
        tmp1*tmp1/(12*(vgfs - vtff - tmp1/2)));
    qd = 0.4*qn;
    qs = 0.6*qn;
```



## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

```
    qgf = width*length*cof*(vgfs - wkf - phi - vds/2 +
        tmp1*vds/ (12*(vgfs - vtff - tmp1/2)));
    qgb = - (qgf + qn + width*length*qb);

end else begin

    //
    // front-channel in saturation.
    //
    id = 0.5*kp*(pow((vgfs - vtff), 2))/(1 + cbb/cof);
    id = id*(1 + lambda*vds);
    qn = -width*length*cof*(2.0/3.0)*(vgfs - vtff);
    qd = 0.4*qn;
    qs = 0.6*qn;
    qgf = width*length*cof*(vgfs - wkf - phi -
        ((vgfs - vtff)/(3*(1 + alpha))));
    qgb = - (qgf + qn + width*length*qb);

end

//
// intrinsic device.
//
I(vdrain, vsource) <+ dev_type_sign*id;
I(vdrain, vgate_back) <+ dev_type_sign*ddt(qd);
I(vsource, vgate_back) <+ dev_type_sign*ddt(qs);
I(vgate_front, vgate_back) <+ dev_type_sign*ddt(qgf);

//
// check back-surface constraints. save the state
// in the back_surf variable. at the final step of
// the analysis, use back_surf to
// print out any possible violations.
//
if (vgbs > vgbi && !back_surf) begin
    back_surf = 1;
end else if (vgbs < vgba && !back_surf) begin
    back_surf = 2;
end

@ ( final_step ) begin
    if (back_surf == 1) begin
        $display("Back-surface went into inversion.\n");
    end else if (back_surf == 2) begin
        $display("Back-surface went into accumulation.\n");
    end
end

end
endmodule
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

The netlist file instantiates an n-channel TFT device with a width of 2 microns ( $2\mu$ ) and a length of 1 micron ( $1\mu$ ). The drain-source voltage ( $v_{ds}$ ) sweeps from 0 to 5 volts.

```
// thin-film transistor example netlist file
//
global gnd
simulator lang=spectre

#define n_type 1

ahdl_include "tft.va"

// Devices
M1_n drain gate source back_gate mos_tft length=1u width=2.5u dev_type=n_type

// Sources
vds drain source vsource dc=5
vbs back_gate source vsource dc=-3
vgs gate_source vsource dc=3

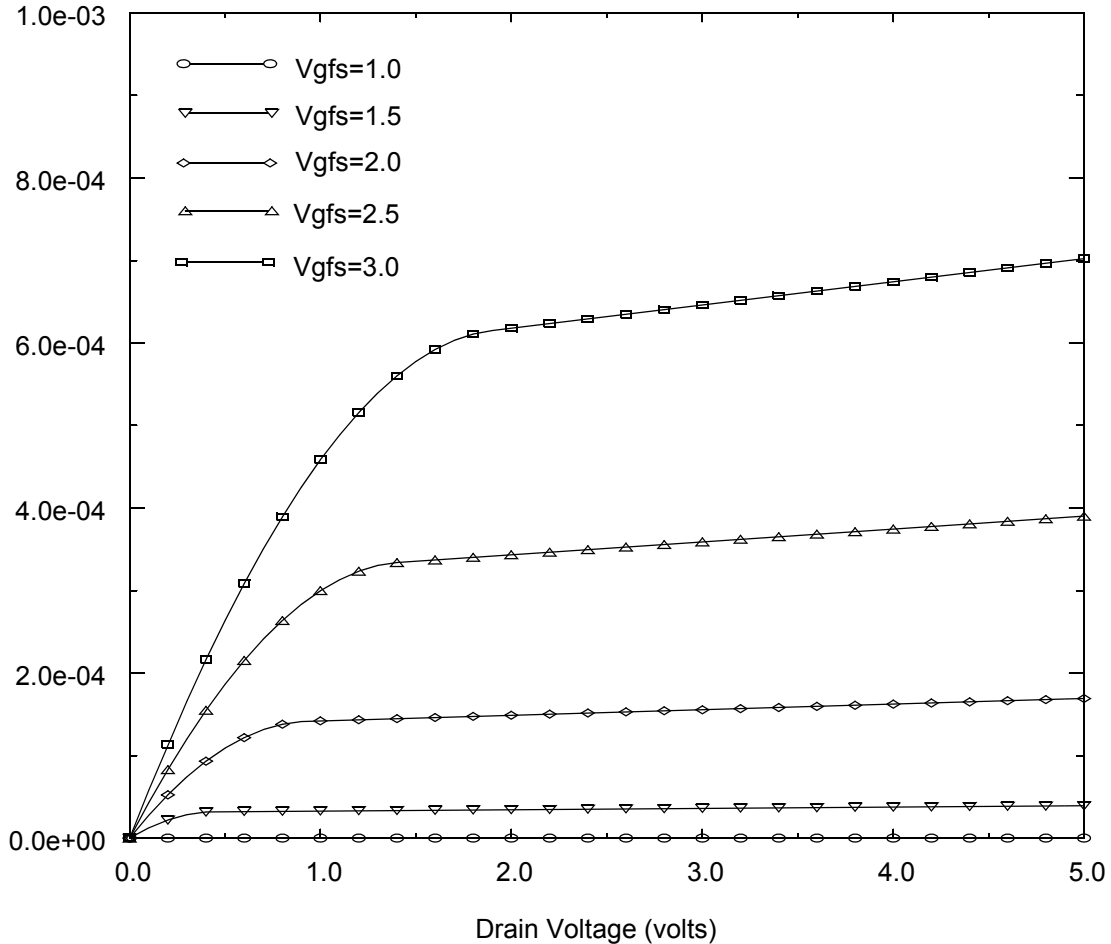
saveOp options save=all currents=all

// Analyses
dcsweep dc start=0 stop=5 step=.1 dev=vds
```

Repeating this sweep for different front gate voltages ( $v_{GS}$ ) with the source gate potential and back gate potential held constant results in the set of I-V characteristics shown in the [I-V Characteristics of the Thin-Film Transistor \(TFT\) Module](#) figure on page 307.

### I-V Characteristics of the Thin-Film Transistor (TFT)

Drain Current (amps)



## Mechanical Modeling

Verilog-A supports multidisciplinary modeling. You can write models representing thermal, chemical, electrical, mechanical, and optical systems and use them together.

This section presents two examples that illustrate the flexibility and power of Verilog-A.

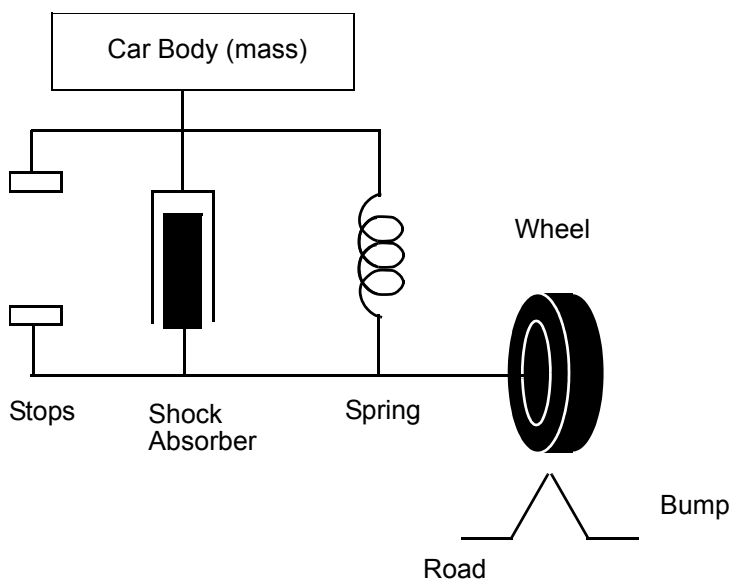
- The first example is a mechanical model of a car wheel on a bumpy road with run-time binding applied to represent the real-world limits of automobile suspensions.

- The second example shows how to create a model of two gears using Verilog-A.

For examples illustrating how Verilog-A can be used to model electrical systems, see [“Electrical Modeling”](#) on page 296.

## Car on a Bumpy Road

This example simulates a car traveling at a fixed speed on a road with a bump in it. This example uses a simple model of a car as a sprung mass.



The equations are formulated with three nodes, one representing the road, one representing the axle, and the third representing the car frame. The potential of each node is its vertical position. The flow out of the nodes is force, which must sum to zero by Kirchhoff's Flow Law.

Verilog-A behavioral descriptions can model the body mass, the spring, the shock absorber, and a triangular shaped bump taken at a particular speed, as well as the car wheel and suspension. The odd mix of units shows how Verilog-A supports arbitrary quantities and units.

### Spring

The spring is a simple linear spring.

```
// spring.va
`include "disciplines.vams"
`include "constants.vams"

module spring (posp, posn);
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

```
inout posp, posn;
kinematic posp, posn;
parameter real k = 5000; // spring constant in lbs/ft
parameter real l = 0.5; // length of spring in feet

    analog
        F(posp,posn) <+ k*(Pos(posp,posn) - l/12.0);

endmodule
```

### Shock Absorber

The shock absorber is a simple linear damper.

```
// damper.va

`include "disciplines.vams"
`include "constants.vams"

module damper (posp, posn);
inout posp, posn;
kinematic posp, posn;
parameter real d = 1000; // friction coef in lbs-s/ft

    analog
        F(posp,posn) <+ d*ddt(Pos(posp,posn));

endmodule
```

### Frame

The frame is modeled as a mass with inertia that is acted on by gravity.

```
// mass.va

`include "disciplines.vams"
`include "constants.vams"

module mass (posin);
inout posin;
kinematic posin;
parameter real m = 1000; // mass given in lbs-mass

    kinematic vel;
    analog begin
        Pos(vel) <+ ddt(Pos(posin));
        F(posin) <+ m*ddt(Pos(vel)/32); // acceleration
        F(posin) <+ m;
    end
endmodule
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

#### Road

The road is modeled as flat, with one or more triangular-shaped obstacles.

The `initial_step` section computes numbers that depend only on input parameters, which is more efficient than doing the calculations in the analog block.

```
// road.va

`include "disciplines.vams"
`include "constants.vams"

module triangle (posin);
inout posin;
kinematic posin;
parameter real height = 4 from (0:inf); // height of bumps (inches)
parameter real width = 12 from (0:inf); // width of bumps (inches)
parameter real speed = 55 from (0:inf); // speed (mph)
parameter real distance = 0 from [0:inf); // distance to first bump (feet)
real duration, offset, Time;

    analog begin

        @ ( initial_step ) begin
            duration = width / (12*1.466667 * speed);
            offset = distance / (1.466667 * speed);
        end

        Time = $abstime - offset;
        if (Time < 0) begin
            Pos(posin) <+ 0;
            @ ( timer( offset ) )
                ; // do nothing, merely place breakpoint
        end else if (Time < duration/2) begin
            Pos(posin) <+ height/6 * Time / duration;
            @ ( timer ( duration / 2 + offset ) )
                ; // do nothing
        end else if (Time < duration) begin
            Pos(posin) <+ height/6 * (1 - Time / duration);
            @ ( timer ( duration + offset ) )
                ; // do nothing
        end else begin
            Pos(posin) <+ 0;
        end
    end
endmodule
```

#### Limiters

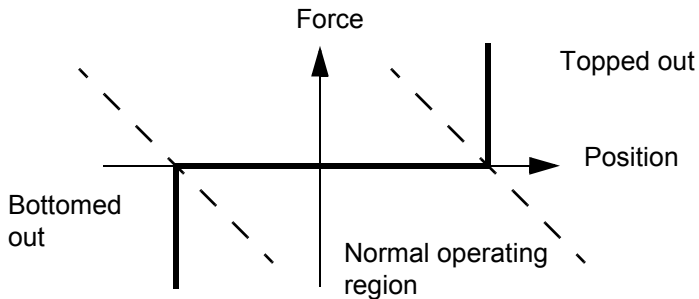
The limiter models the limited travel of an automotive suspension using the run time binding of potential and flow sources to implement the mechanical constraints (the stops) in the suspension.

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

The limiter keeps the distance between two points inside a certain range by placing a rigid constraint on the distance. However, within the range, the limiter has no effect. A plot of force versus position is as follows.



This model uses length to determine which region the limiter is in. If the length is less than `maxl` and greater than `minl`, the model must be in the normal operating region. If the length is less than or equal to `minl`, the limiter has bottomed out. However, because of the limiting, the length cannot be less than `minl`, so the limiter bottoms out if the length equals `minl`. This is a dangerous test. Any error in the calculation causes the limiter to jump back and forth from the normal region to being bottomed out. The model is abruptly discontinuous at the region boundaries.

Continually crossing from one region to another causes the simulator to run slowly and can create convergence difficulties. For this reason, the region boundaries used are those given by the dotted lines in the figure. Both position and force are taken into account when determining which region the limiter is in. This is a much more reliable method for determining the operating region of the limiter.

```
// limiter.va

`include "disciplines.vams"
`include "constants.vams"

module limiter (posp, posn);
  inout posp, posn;
  kinematic posp, posn;
  parameter real minl = 2; // minimum extension in inches
  parameter real maxl = 10; // maximum extension in inches
  integer out_of_range;
  integer too_long, too_short;

  analog begin
    if (Pos(posp,posn) - maxl/12 + F(posp,posn) / 10.0e3 > 0.0) begin
      Pos(posp,posn) <+ maxl/12;
      too_long = 1;
      too_short = 0;
    end else if (Pos(posp,posn) - minl/12 + F(posp,posn) / 10.0e3 < 0.0) begin
      Pos(posp,posn) <+ minl/12;
      too_long = 0;
      too_short = 1;
    end else begin
      F(posp,posn) <+ 0;
    end
  end
endmodule
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

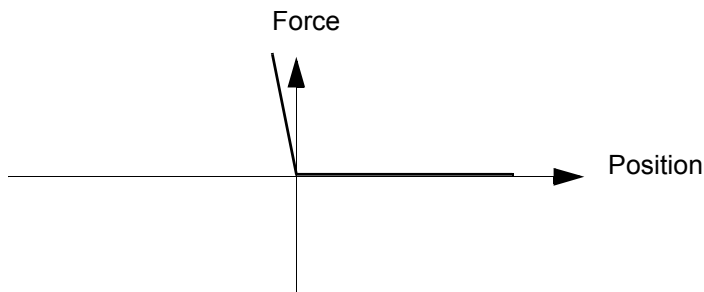
```
        too_long = 0;
        too_short = 0;
    end
    if (out_of_range) begin
        if (!too_long && !too_short) begin
            out_of_range = 0;
            $strobe( "%M: In range again at t = %E s.\n", $abstime );
        end
    end else begin
        if (too_long) begin
            $strobe( "%M: Topped out at t = %E s.\n", $abstime );
            out_of_range = 1;
        end
        else if (too_short) begin
            $strobe( "%M: Bottomed out at t = %E s.\n", $abstime );
            out_of_range = 1;
        end
    end
end
end
endmodule
```

When the limiter changes from one region to another, the simulator prints messages.

This module can be difficult to debug because it is abruptly discontinuous. One approach to this problem is to reduce the strength of the module by putting a small resistor in series with the limiter. The resistor lets the Spectre<sup>®</sup> circuit simulator converge, so you can use the normal printing and plotting aids for debugging. Once the limiter is behaving properly, you can remove the resistor.

### Wheel

The important effect being modeled with the wheel is that it can lift off the ground. Dynamic binding is used to model the fact that the wheel can push on the ground, but it cannot pull. In addition, the elasticity of the wheel is modeled. The force-versus-position characteristics of the wheel are shown with the module definition as follows.



```
// wheel.va
`include "disciplines.vams"
`include "constants.vams"

module wheel (posp, posn);
```



## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

```
inout posp, posn;
kinematic posp, posn;
parameter real height = 0.5 from (0:inf);
integer reported;
integer flying;

    analog begin
        if (Pos(posp, posn) < height) begin
            Pos(posp, posn) <+ height + F(posp, posn) / 200K;
            flying = 0;
        end else begin
            F(posp, posn) <+ 0;
            flying = 1;
        end
        if (reported) begin
            if (!flying) begin
                reported = 0;
                $strobe( "%M: On ground again at t = %E s.\n", $abstime );
            end
        end else begin
            if (flying) begin
                $strobe( "%M: Airborne at t = %E s.\n", $abstime );
                reported = 1;
            end
        end
    end
end
endmodule
```

### The System

Two nodes are used to model the automobile, one for the frame and one for the axle. Another node is used to model the surface of the road. The potential of all three nodes is the vertical position, with up being positive. The flow at the nodes is force, with upward forces being positive.

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

The car is driven over 1-, 3-, and 6-inch triangular obstacles at 55 miles per hour. The vertical position of the frame, axle, and road and the force on the road are plotted versus time for the 6-inch obstacle.

```
// netlist for Car on bumpy road
simulator lang=spectre
spectre options quantities=full save=all

// include Verilog-A models
ahdl_include "mass.va"
ahdl_include "spring.va"
ahdl_include "limiter.va"
ahdl_include "damper.va"
ahdl_include "wheel.va"
ahdl_include "road.va"

// describe sprung mass on bumpy road
Body   frame      mass m=2.5klbs
Spring frame axle spring k=5k l=9
Shock  frame axle damper d=700
Stops  frame axle limiter minl=1 maxl=5
Wheel  axle road  wheel
Bump   road      triangle height=1_in width=24_in speed=55_mph

nodeset frame=0 axle=0

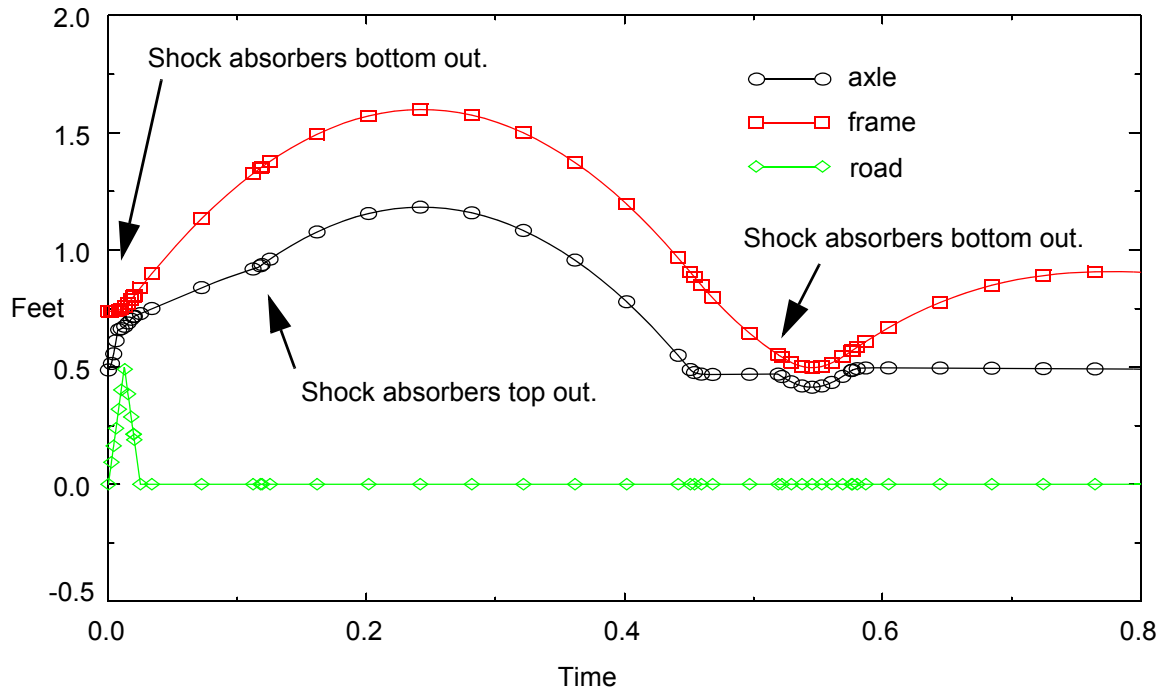
// perform transient analysis
bump tran stop=1 errpreset=conservative
higher alter dev=Bump param=height value=3_in
whack tran stop=1 errpreset=conservative
andLarger alter dev=Bump param=height value=6_in
launch tran stop=1 errpreset=conservative
```

During the simulation of the 6-inch obstacle, the Spectre simulator prints results that contain messages from the limiter and the wheel that indicate when they changed regions.

```
*****
Transient Analysis `launch': time=(0 s -> 1 s)
*****

Stops: Bottomed out at t = 7.292152e-03 s.
Stops: In range again at t = 1.941606e-02 s.
Wheel: Airborne at t = 1.957681e-02 s.
Stops: Topped out at t = 1.163974e-01 s.
Wheel: On ground again at t = 4.493263e-01 s.
Stops: In range again at t = 4.507094e-01 s.
Stops: Bottomed out at t = 5.197922e-01 s.
Stops: In range again at t = 5.755469e-01 s.
```

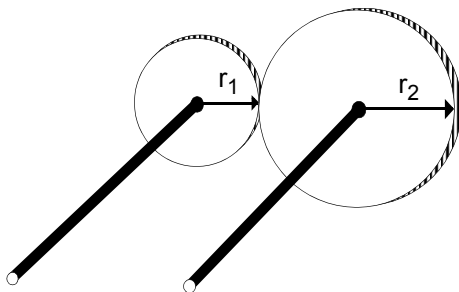
### Transient Response in Car on a Bumpy Road



Looking at this plot, you can visualize the car flying into the air, with its wheels drooping below it, then the wheels and the car slamming into the ground. The weight of the car flattens the tires at 0.55 seconds.

### Gearbox

This Verilog-A module models a gearbox that consists of two shafts and two gears. The model is bidirectional, meaning that either shaft can be driven, and the loading is passed from the driven shaft to the driving shaft. Inertia in each gear and shaft is also modeled.



In this example, you choose the variables with which to formulate the model. Then you develop the constitutive relationships and convert the constitutive relationships into a Verilog-A module.

### Choosing the Variables

The gearbox connects to the rest of the system through shafts. A module connects to the rest of a network through terminals. Here the module is formulated with the shafts as the terminals of the module. The important quantities of the shafts are their angular velocities (frequency) and the torques they exert on the rest of the system. Both quantities (frequency and torque) are associated with each shaft. In this case, angular velocity or frequency is the natural choice for potential because it satisfies Kirchhoff's Potential Law. Angular velocity must satisfy Kirchhoff's Potential Law because it is the derivative of angular position, which clearly satisfies Kirchhoff's Potential Law (a complete rotation sums to zero). Torque is the natural choice for flow because it satisfies Kirchhoff's Flow Law.

### Choosing the Reference Directions

Torque is considered positive if it accelerates a gear in a counterclockwise direction. Likewise, angular velocity is positive in the counterclockwise direction. Torque (the flow) is taken to be positive if it flows from outside the module, through the shaft, into the gearbox. In this example, both frequency and torque are specified in absolute terms, meaning that all measurements are relative to ground (the resting state).

### The Physics

There are three sources of torque on each shaft:

- The torque applied externally through the shaft
- The torque applied from the other gear through the teeth of the gear on the shaft
- The torque needed to accelerate the inertia of the shaft and gear

These torques must balance:

$$\tau_{ext} + \tau_{teeth} + \tau_{inertia} = 0$$

or

$$\tau_{ext} + rF_{teeth} + I\alpha = 0$$

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

where  $r$  is the radius of the gear,  $I$  is the inertia of the gear and shaft, and  $\alpha$  is the angular acceleration. The angular acceleration is given by

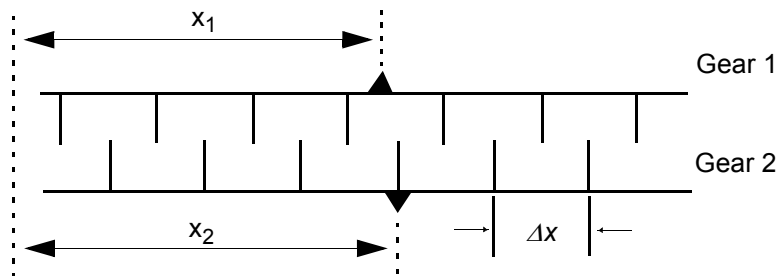
$$\alpha = \frac{d\omega}{dt}$$

$$\omega = \frac{d\theta}{dt}$$

where  $\omega$  is the angular velocity and  $\theta$  is the angular position or phase of the shaft.

To simplify the development of the model, assume that the gears and shaft have no inertia.

To show the interaction of the two gears, the following figure peels the gear teeth from the circular gear and flattens them. This allows the equations to be formulated in rectangular coordinates.



The translational position of the gear teeth is related to the angular position of the gear by

$$x_1 = 2\pi r_1 \theta_1$$

Because gear 2 rotates backwards

$$x_2 = -2\pi r_2 \theta_2$$

Assume that the teeth mesh perfectly, so that the gearbox does not exhibit backlash. Then the positions of both gears must match.

$$x_1 = x_2$$

or

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

$$2\pi r_1 \theta_1 = -2\pi r_2 \theta_2$$

This can be rewritten to explicitly give  $\theta_1$  in terms of  $\theta_2$ .

$$\theta_1 = -\frac{r_2}{r_1} \theta_2 \quad (\text{phase})$$

The torque on the shaft due to the interaction of the teeth can be computed from the force at the teeth with

$$\tau = rF$$

At the point of contact of the two gears, the forces must balance

$$F_1 = -F_2$$

or

$$(14-1) \quad \frac{\hat{\tau}_1}{r_1} = \frac{\hat{\tau}_2}{r_2}$$

where  $\tau_1$  and  $\tau_2$  are the torques applied to the shafts by the external system, assuming that the gear and shaft have no inertia.

$$\hat{\tau}_2 = \frac{r_2}{r_1} \hat{\tau}_1 \quad (\text{torque})$$

Finally, the effect of the inertia of the gear and shaft is added.

$$\tau = \hat{\tau} + I\alpha$$

where  $\tau$  is the total torque applied externally to the shaft,  $\hat{\tau}$  is the torque used to push the other gear, and  $I\alpha$  is the torque required to accelerate the inertia of the shaft and gear. The torque equation can now be rewritten to include the effect of inertia:

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

---

$$\tau_2 = I_2 \alpha_2 - \frac{r_2}{r_1} (\tau_1 - I_1 \alpha_1) \quad (\text{full torque})$$

### Implementation of the Gearbox Model

The phase and full torque equations are the constitutive equations for the gearbox. The natures for velocity ( $\omega$ ) and torque ( $\tau$ ) are defined in the `disciplines.vams` file.

```
// gearbox.va

`include "disciplines.vams"
`include "constants.vams"

module gearbox(wshaft1, wshaft2);
inout wshaft1, wshaft2;
rotational_omega wshaft1, wshaft2;
parameter real radius1=1 from (0:inf);
parameter real inertia1=0 from [0:inf);
parameter real radius2=1 from (0:inf);
parameter real inertia2=0 from [0:inf);

    analog begin

        //
        // Calculate the angular velocity of shaft1 from
        // that of shaft2
        //
        Omega( wshaft1 ) <+ Omega( wshaft2 ) * radius2 / radius1;

        // Calculate the torque on shaft1 from the torque
        // on shaft2 and the angular acceleration.
        //
        Tau( wshaft2 ) <+ inertia2 * ddt( Omega( wshaft2 ) )
            + (Tau( wshaft1 ) - inertia1 *
              ddt( Omega( wshaft1 ) ))
            * (radius2 / radius1);

    end
endmodule
```

## Cadence Verilog-A Language Reference

### Verilog-A Modeling Examples

A system constructed from Spectre simulator primitives quickly tests this module. A current source and resistor model a motor, and a resistor models a load. The rotational nodes,  $s_1$  and  $s_2$ , represent shafts.

```
// Gearbox test system netlist file
simulator lang=spectre

ahdl_include "gearbox.va"

P1 s1 0 isource type=pwl wave=[0 0 1 1]
P2 s1 0 resistor r=1
GB1 s1 s2 gearbox radius1=2 inertial1=0.2 inertia2=0.1
L1 s2 0 resistor r=1

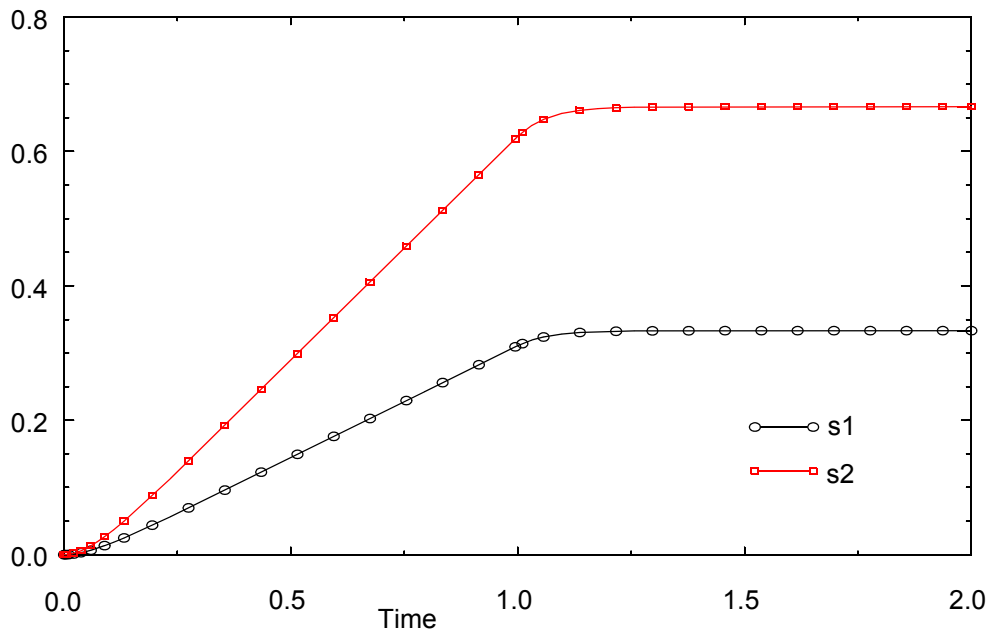
timeResp tran stop=2

modifyOmega quantity name="Omega" abstol=1e-4
modifyTau quantity name="Tau" abstol=1e-4
```

The motor drives the gearbox with a finite slope step change in torque.

### Transient Response of the Gearbox

Rotational Velocity





## Computing a Moving or Sliding-Window Average

You can write Verilog-A code to compute the moving or sliding-window average of an input signal as follows:

```
// VerilogA for MyLib, moving_average, veriloga
//
// Calculates the moving average (or: sliding-window average)
// of the input signal x(t):
//
//      gain      t
// y(t) = IC + ---- * Integral{x(tau)dtau}
//      window    t-window
//
// Copyright (c) 2006
// by Cadence Design Systems, Inc. All rights reserved.
//
// Written by Ihor Harasymiv, 02.10.2006

`include "constants.vams"
`include "disciplines.vams"

module moving_average(in, out);
input in;
output out;
voltage in, out;

parameter real gain=1.0;
parameter real time_window= 1u from (0:inf);

real x, k1;

analog begin
  @(initial_step) k1 = gain/time_window;

  x= V(in)- absdelay(V(in),time_window);
  V(out) <+ k1*idt(x,0.0);
end
```

**Cadence Verilog-A Language Reference**  
Verilog-A Modeling Examples

---

---

## Nodal Analysis

---

This appendix briefly introduces Kirchhoff's Laws and describes how the simulator uses them to simulate a system. For information, see

- [Kirchhoff's Laws](#) on page 324
- [Simulating a System](#) on page 325

## Kirchhoff's Laws

Simulation of Verilog<sup>®</sup>-A language modules is based on two sets of relationships. The first set, called the *constitutive relationships*, consists of formulas that describe the behavior of each component. Some formulas are supplied as built-in primitives. You provide other formulas in the form of module definitions.

The second set of relationships, the *interconnection relationships*, describes the structure of the network. This set, which contains information on how the nodes of the components are connected, is independent of the behavior of the constituent components. Kirchhoff's laws provide the following properties relating the quantities present on the nodes and on the branches that connect the nodes.

- Kirchhoff's Flow Law

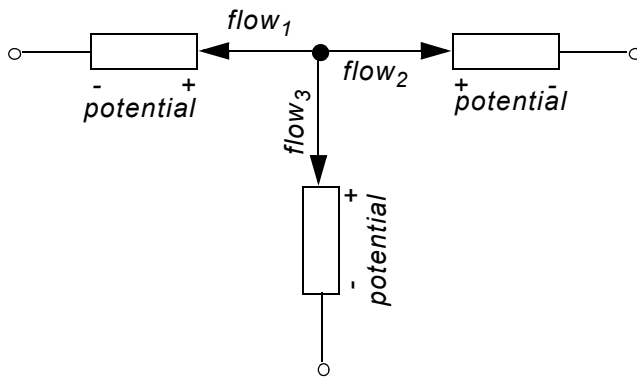
The algebraic sum of all the flows out of a node at any instant is zero.

- Kirchhoff's Potential Law

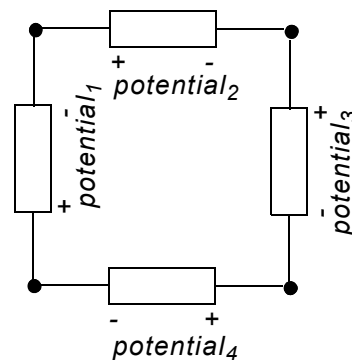
The algebraic sum of all the branch potentials around a loop at any instant is zero.

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

### Kirchhoff's Laws



$$flow_1 + flow_2 + flow_3 = 0$$



$$potential_1 + potential_2 + potential_3 + potential_4 = 0$$

## Simulating a System

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.

$v$  is a vector containing all node potentials.

$t$  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.

## Transient Analysis

The equation describing the network is differential and nonlinear, which makes it impossible to solve directly. There are a number of different approaches to solving this problem numerically. However, all approaches break time into increments and solve the nonlinear equations iteratively.

The simulator replaces the time derivative operator ( $\partial q / \partial t$ ) 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, the simulator solves iteratively a system of nonlinear algebraic equations. Like most circuit simulators, the Spectre uses the Newton-Raphson method to solve this system.

## Convergence

In Verilog-A, the behavioral description is evaluated iteratively until the Newton-Raphson method converges. (For a graphical representation of this process, see [“Simulator Flow”](#) on page 31.) On the first iteration, the signal values used in Verilog-A expressions are approximate and do not satisfy Kirchhoff's laws.

## Cadence Verilog-A Language Reference

### Nodal Analysis

---

In fact, the initial values might not be reasonable; so you must write models that do something reasonable even when given unreasonable signal values.

For example, if you compute the log or square root of a signal value, 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 that the proposed solution on this iteration,  $v_n^{(j)}(t)$ , must be close to the proposed solution on the previous iteration,  $v_n^{(j-1)}(t)$ , and

$$\left| v_n^{(j)} - v_n^{(j-1)} \right| < reltol \left( \max \left( \left| v_n^{(j)} \right|, \left| v_n^{(j-1)} \right| \right) \right) + abstol$$

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, and which one is used depends on the resolved discipline of the net. You set absolute tolerances by specifying the *abstol* attribute for the natures you use. The absolute tolerance is important when  $v_n$  is converging to zero. Without *abstol*, the iteration never converges.

The second criterion ensures that Kirchhoff's Flow Law is satisfied:

$$\left| \sum_n f_n^i(v^{(j)}) \right| < reltol \left( \max \left( \left| f_n^i(v^{(j)}) \right| \right) \right) + abstol$$

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 that convergence is not precluded when  $v_n$  or  $f_n(v)$  go to zero. While you can set the relative tolerance once in an options statement to work effectively on any node in the circuit, you must scale the absolute tolerance appropriately for the associated branches. Set the absolute tolerance to be the largest value that is negligible on all the branches with which 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 1  $\mu$ V. The largest current is about 1 mA; so the default absolute tolerance for current is 1 pA.

---

## Analog Probes and Sources

---

This appendix describes what analog probes and sources are and gives some examples of using them. For information, see

- [Probes](#) on page 328
- [Sources](#) on page 329

For examples, see

- [Linear Conductor](#) on page 334
- [Linear Resistor](#) on page 335
- [RLC Circuit](#) on page 335
- [Simple Implicit Diode](#) on page 335

## Overview of Probes and Sources

A *probe* is a branch in which no value is assigned for either the potential or the flow, anywhere in the module. A *source* is a branch in which either the potential or the flow is assigned a value by a contribution statement somewhere in the module.

You might find it useful to describe component behavior as a network of probes and sources.

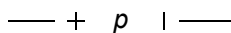
- It is sometimes easier to describe a component first as a network of probes and sources, and then use the rules presented here to map the network into a behavioral description.
- A complex behavioral description is sometimes easier to understand if it is converted into a network of probes and sources.

The probe and source interpretation provides the additional benefit of unambiguously defining what the response will be when you manipulate a signal.

## Probes

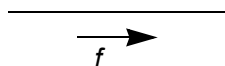
A *flow probe* is a branch in which the flow is used in an expression somewhere in the module. A *potential probe* is a branch in which the potential is used. You must not measure both the potential and the flow of a probe branch.

The equivalent circuit model for a potential probe is



The branch flow of a potential probe is zero.

The equivalent circuit model for a flow probe is



The branch potential of a flow probe is zero.

## Port Branches

A port branch, which is a special form of a flow probe, measures the flow into a port rather than across a branch. When a port is connected to numerous branches, using a port branch provides a quick way of summing the flow.



## Cadence Verilog-A Language Reference

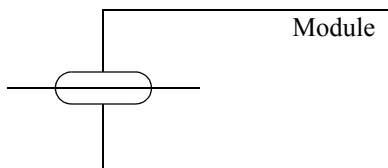
### Analog Probes and Sources

---

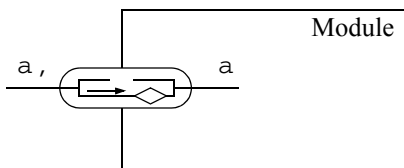
You can use the port access function to monitor the flow into the port of a module. The name of the access function is derived from the flow nature of the discipline of the port and you use the (<>) operator to delimit the port name. For example, `I(<a>)` accesses the current through module port `a`.

The figure below illustrates the difference between a port branch and a simple port:

#### Simple port



#### Port branch



In the simple port, the two sides of the port are indistinguishable. In the port branch, the two terminals of the port, `a'` and `a`, are distinguishable, so that a flow probe can be implemented across them. Establishing a flow probe is all you can do with a port branch—you cannot set the flow, nor can you read or set the potential.

You can use a port branch to monitor the flow. In the following example, the simulator issues a warning if the current through the `a` port branch becomes too large.

```
module diode (a, c) ;
  electrical a, c ;
  branch (a, c) diode, cap ;
  parameter real is=1e-14, tf=0, cjo=0, imax=1, phi=0.7 ;

  analog begin
    I(diode) <+ is*($limexp(V(diode)/$vt) - 1) ;
    I(cap) <+ ddt(tf*I(diode) - 2 * cjo *
      sqrt(phi * (phi * V(cap)))) ;
    if (I(<a>) > imax) // Checks current through port
      $strobe( "Warning: diode is melting!" ) ;
  end
endmodule
```

## Sources

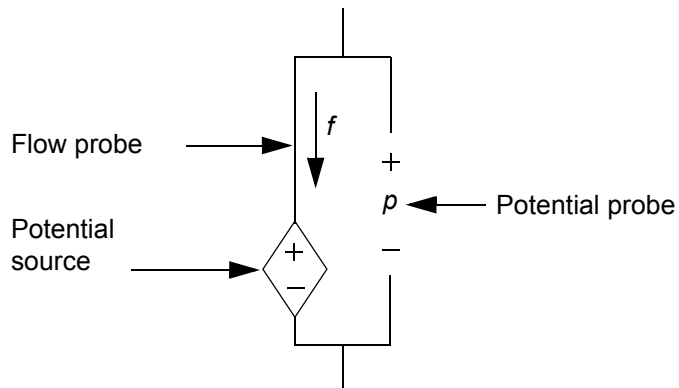
A *potential source* is a branch in which the potential is assigned a value by a contribution statement somewhere in the module. A *flow source* is a branch in which the flow is assigned a value. A branch cannot simultaneously be both a potential and a flow source, although it can switch between the two kinds. For additional information, see [“Switch Branches”](#) on page 331.

## Cadence Verilog-A Language Reference

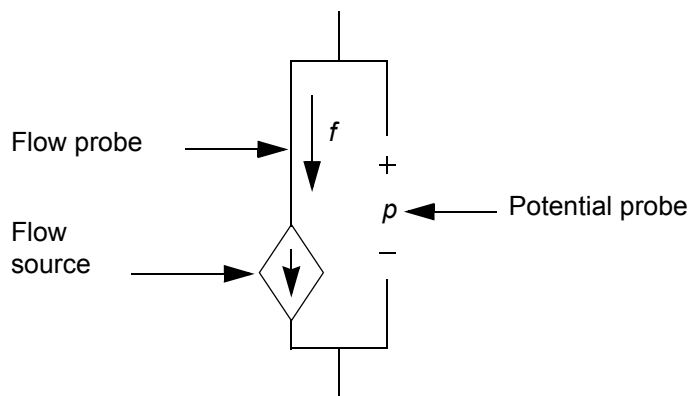
### Analog Probes and Sources

---

The circuit model for a potential source branch shows that you can obtain both the flow and the potential for a potential source branch.



Similarly, the circuit model for a flow source branch shows that you can obtain the flow and potential for a flow source branch.



With the flow and potential sources, you can model the four basic controlled sources, using node or branch declarations and contribution statements like those in the following code fragments.

The model for a *voltage-controlled voltage source* is

```
branch (ps,ns) in, (p,n) out;  
V(out) <+ A * V(in);
```

The model for a *voltage-controlled current source* is

```
branch (ps,ns) in, (p,n) out;  
I(out) <+ A * V(in);
```

The model for a *current-controlled voltage source* is

```
branch (ps,ns) in, (p,n) out;  
V(out) <+ A * I(in);
```

The model for a *current-controlled current source* is

```
branch (ps,ns) in, (p,n) out;  
I(out) <+ A * I(in);
```

## Unassigned Sources

If you do not assign a value to a branch, the branch flow, by default, is set to zero. In the following fragment, for example, when `closed` is true,  $V(p,n)$  is set to zero. When `closed` is false, the current  $I(p,n)$  is set to zero.

```
if (closed)  
    V(p,n) <+ 0 ;  
else  
    I(p,n) <+ 0 ;
```

Alternatively, you could achieve the same result with

```
if (closed)  
    V(p,n) <+ 0 ;
```

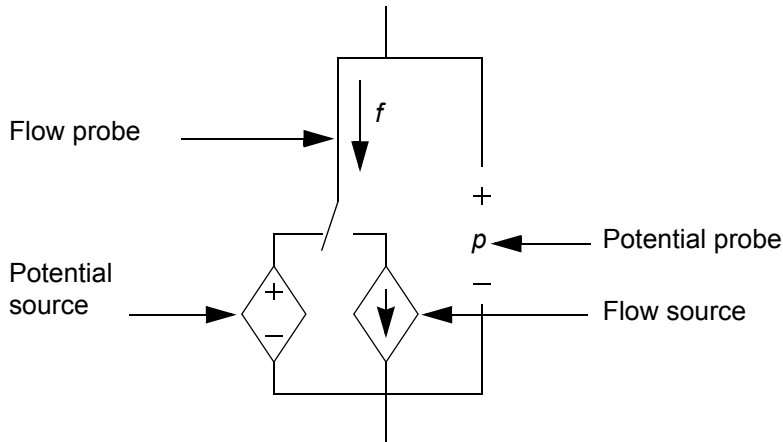
This code fragment also sets  $V(p,n)$  to zero when `closed` is true. When `closed` is false, the current is set to zero by default.

## Switch Branches

*Switch branches* are branches that change from source potential branches into source flow branches, and vice versa. Switch branches are useful when you want to model ideal switches or mechanical stops.

To switch a branch to being a potential source, assign to its potential. To switch a branch to being a flow source, assign to its flow. The circuit model for a switch branch illustrates the

effect, with the position of the switch dependent upon whether you assign to the potential or to the flow of the branch.



As an example of a switch branch, consider the module `idealRelay`.

```

module idealRelay (pout, nout, psense, nsense) ;
input psense, nsense ;
output pout, nout ;
electrical pout, nout, psense, nsense ;
parameter real thresh = 2.5 ;
analog begin
    if (V(psense, nsense) > thresh)
        V(pout, nout) <+ 0.0 ; // Becomes potential source
    else
        I(pout, nout) <+ 0.0 ; // Becomes flow source
    end
endmodule

```

The simulator assumes that a discontinuity of order zero occurs whenever the branch switches; so you do not have to use the discontinuity function with switch branches. For more information about the discontinuity function, see [“Announcing Discontinuity”](#) on page 127.

Contributing a flow to a branch that 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 that already has a value retained for the flow results in the flow being discarded and the branch being converted to a potential source. For example, in the following code, each of the contribution statements is discarded when the next is encountered.

```

analog begin
    V(out) <+ 1.0; // Discarded
    I(out) <+ 1.0; // Discarded
    V(out) <+ 1.0;
end

```

In the next example,

## Cadence Verilog-A Language Reference

### Analog Probes and Sources

---

```
I(out) <+ 1.0;
V(out) <+ I(out);
```

the result of `V(out)` is not 1.0. Instead, these two statements are equivalent to

```
// I(out) <+ 1.0;
V(out) <+ I(out);
```

because the flow contribution is discarded. The simulator reminds you of this behavior by issuing a warning similar to the following,

```
The statement on line 12 contributes either a potential to a flow source or a flow
to a potential source. To match the requirements of value retention, the statement
is ignored.
```

### Troubleshooting Loops of Rigid Branches

The following message might not actually indicate an error in your code.

```
Fatal error found by spectre during topology check.
The following branches form a loop of rigid branches (shorts)...
```

Sometimes the simulator takes a too conservative approach to checking switch branches by assuming, when it is not actually the case, that all switch branches are in the voltage source mode at the same time. To disable this assumption, you can use the Cadence `no_rigid_switch_branch` attribute. To avoid convergence difficulties, however, do not use this attribute when you really do have multiple voltage sources in parallel or current sources in series.

To illustrate how the `no_rigid_switch_branch` can be used, assume that you have the following module.

```
// Verilog-A for sourceSwitch
`include "constants.vams"
`include "discipline.vams"
module sourceSwitch(vip1, vin1, vip2, vin2, vop1, von1);
    input vip1, vin1, vip2, vin2;
    output vop1, von1;
    electrical vip1, vin1, vip2, vin2, vop1, von1;
    parameter integer swState = 0;
//    (* no_rigid_switch_branch *) analog
    analog //this block causes a topology check error
    begin
        if ( swState == 0 )
            begin
                V(vop1, vip1) <+ 1.0;
                V(von1, vin1) <+ 1.0;
            end
        else if (swState == 1 )
            begin
                V(vop1, vip2) <+ 1.0;
                V(von1, vin2) <+ 1.0;
            end
    end
end
```

## Cadence Verilog-A Language Reference

### Analog Probes and Sources

---

```
end  
endmodule
```

Attempting to run this module produces the following error:

```
Fatal error found by spectre during topology check.  
The following branches form a loop of rigid branches (shorts) when  
added to the circuit:  
v1:p (from vip1 to 0)  
myswitch:von1_vin2_flow (from von1 to 0)
```

In this example, you can use the `no_rigid_switch_branch` attribute to turn off the checking because the check indicates a problem when there actually is no problem. To use the attribute, you insert it before the analog block. (In the illustrated module, you can just uncomment the row containing the `no_rigid_switch_branch` attribute and comment out the following row.)

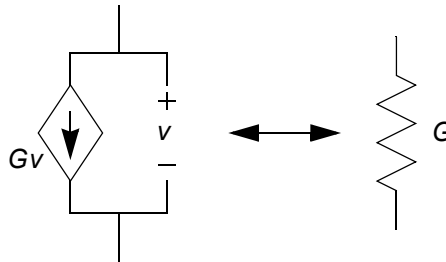
## Examples of Sources and Probes

The following examples illustrate how to construct models using sources and probes.

### Linear Conductor

The model for a linear conductor is

```
Module myconductor(p,n) ;  
parameter real G=1 ;  
electrical p,n ;  
branch (p,n) cond ;  
analog begin  
    I(cond) <+ G * V(cond);  
end  
endmodule
```



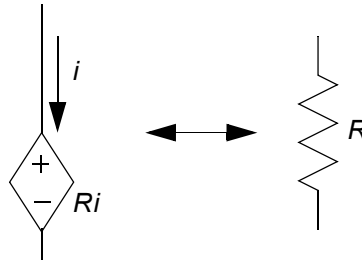
The contribution to `I(cond)` makes `cond` a current (flow) source branch, and `V(cond)` accesses the potential probe built into the current source branch.

## Linear Resistor

The model for a linear resistor is

```

module myresistor(p,n) ;
parameter real R=1 ;
electrical p,n;
branch (p,n) res ;
analog begin
    V(res) <+ R * I(res);
end
endmodule
    
```



The contribution to  $V(res)$  makes  $res$  a potential source branch.  $I(res)$  accesses the flow probe built into the potential source branch.

## RLC Circuit

A series RLC circuit is formulated by summing the voltage across the three components.

$$v(t) = Ri(t) + L\frac{d}{dt}i(t) + \frac{1}{C}\int_{-\infty}^t i(\tau)d\tau$$

To describe the series RLC circuit with probes and sources, you might write

```
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 the three components.

$$i(t) = \frac{v(t)}{R} + C\frac{d}{dt}v(t) + \frac{1}{L}\int_{-\infty}^t v(\tau)d\tau$$

To describe the parallel RLC circuit, you might code

```
I(p,n) <+ V(p,n)/R + C*ddt(V(p,n)) + idt(V(p,n))/L ;
```

## Simple Implicit Diode

This example illustrates a case where the model equation is implicit. The model equation is implicit because the current  $I(a,c)$  appears on both sides of the contribution operator. The equation specifies the current of the branch, making it a flow source branch. In addition, both the voltage and the current of the branch are used in the behavioral description.

```
I(a,c) <+ is * (limexp((V(a,c) - rs * I(a,c)) / Vt) - 1) ;
```

**Cadence Verilog-A Language Reference**  
Analog Probes and Sources

---



---

## Sample Model Library

---

This appendix discusses the Sample Model Library, which is included with this product. The library contains the following types of components:

- [Analog Components](#) on page 339
- [Basic Components](#) on page 356
- [Control Components](#) on page 364
- [Logic Components](#) on page 372
- [Electromagnetic Components](#) on page 392
- [Functional Blocks](#) on page 395
- [Magnetic Components](#) on page 418
- [Mathematical Components](#) on page 422
- [Measure Components](#) on page 439
- [Mechanical Systems](#) on page 459
- [Mixed-Signal Components](#) on page 466
- [Power Electronics Components](#) on page 475
- [Semiconductor Components](#) on page 478
- [Telecommunications Components](#) on page 486

You can use these models as they are, you can copy them and modify them to create new parts, or you can use them as examples. The models are in the following directory in the software hierarchy:

```
your_install_dir/tools/dfII/samples/artist/spectreHDL/Verilog-A
```

Refer to the README file in this directory for a list of the files containing the models. The filenames have the suffix `.va`. For example, the model for the switch is located in `sw.va`. Each model has an associated test circuit that can be used to simulate the model.

## Cadence Verilog-A Language Reference

### Sample Model Library

---

These models are also integrated into a Cadence® design framework II library, complete with symbols and Component Description Formats (CDFs). If you are using the Cadence analog design environment, you can access these models by adding the following library to your library path:

```
your_install_dir/tools/dfII/samples/artist/ahdlLib
```

This appendix provides a list of the parts and functions in the sample library. They are grouped according to application.

In the terminal description and parameter descriptions, the letters between the square brackets, such as [V,A] and [V], refer to the units associated with the terminal or parameter. V means volts, A means amps. (val, flow) means that any units can be used.

## Analog Components

### Analog Multiplexer

#### Terminals

`vin1, vin2:` [V,A]  
`vsel:` selection voltage [V,A]  
`vout:` [V,A]

#### Description

When `vsel > vth`, the output voltage follows `vin1`.

When `vsel < vth`, the output voltage follows `vin2`.

#### Instance Parameters

`vth = 1->0` threshold voltage for the selection line [V]

## Current Deadband Amplifier

### Terminals

`iin_p, iin_n`: differential input current terminals [V,A]

`iout`: output current terminal [V,A]

### Description

Outputs `ileak` when differential input current (`iin_p - iin_n`) is between `idead_low` and `idead_high`. When outside the deadband, the output current is an amplified version of the differential input current plus `ileak`.

### Instance Parameters

`idead_low` = lower range of dead band [A]

`idead_high` = upper range of dead band [A]

`ileak` = offset current; only output in deadband [A]

`gain_low` = differential current gain in lower region []

`gain_high` = differential current gain in lower region []

## Hard Current Clamp

### Terminals

`vin:` input terminal [V,A]

`vout:` output terminal [V,A]

`vgn:` gnd terminal [V,A]

### Description

Hard limits output current to between `iclamp_upper` and `iclamp_lower` of the input current.

### Instance Parameters

`iclamp_upper` = upper clamping current [A]

`iclamp_lower` = lower clamping current [A]

## Hard Voltage Clamp

### Terminals

`vin`: input terminal [V,A]

`vout`: output terminal [V,A]

`vgn`: gnd terminal [V,A]

### Description

`vout-vgn` hard clamped/limited to between `vclamp_upper` and `vclamp_lower` of `vin-vgn`.

### Instance Parameters

`vclamp_upper` = upper clamping voltage [A]

`vclamp_lower` = lower clamping voltage [A]

## Open Circuit Fault

### Terminals

`vp`, `vn`: output terminals [V,A]

### Description

At time=`twait`, the connection between the two terminals is opened. Before this, the connection between the terminals is closed.

### Instance Parameters

`twait` = time to wait before open fault happens [s]

## Operational Amplifier

### Terminals

<code>vin_p, vin_n:</code>	differential input voltage [V,A]
<code>vout:</code>	output voltage [V,A]
<code>vref:</code>	reference voltage [V,A]
<code>vsupply_p:</code>	positive supply voltage [V,A]
<code>vsupply_n:</code>	negative supply voltage [V,A]

### Instance Parameters

<code>gain = gain []</code>
<code>freq_unitygain = unity gain frequency [Hz]</code>
<code>rin = input resistance [Ohms]</code>
<code>vin_offset = input offset voltage referred to negative [V]</code>
<code>ibias = input current [A]</code>
<code>iin_max = maximum current [A]</code>
<code>rsrc = source resistance [Ohms]</code>
<code>rout = output resistance [Ohms]</code>
<code>vsoft = soft output limiting value [V]</code>



## Constant Power Sink

### Terminals

`vp`, `vn`: terminals [V,A]

### Description

Normally `power` watts of power is sunk. If the absolute value of `vp - vn` is above `vabsmin`, a fraction of the `power` is sunk. The fraction is the ratio of `vp - vn` to `vabsmin`.

### Instance Parameters

`power` = power sunk [Watts]

`vabsmin` = absolute value of minimum input voltage [V]

## Short Circuit Fault

### Terminals

`vp`, `vn`: output terminals [V,A]

### Description

At time=`twait`, the two terminals short. Before this, the connection between the terminals is open.

### Instance Parameters

`twait` = time to wait before short circuit occurs [s]

## Soft Current Clamp

### Terminals

`vin`: input terminal [V,A]  
`vout`: output terminal [V,A]  
`vgnd`: gnd terminal [V,A]

### Description

Limits output current to between `iclamp_upper` and `iclamp_lower` of the input current.

The limiting starts working once the input current gets near `iclamp_lower` or `iclamp_upper`. The clamping acts exponentially to ensure smoothness.

The fraction of the range (`iclamp_lower`, `iclamp_upper`) over which the exponential clamping action occurs is `exp_frac`.

Excess current coming from `vin` is routed to `vgnd`.

### Instance Parameters

`iclamp_upper` = upper clamping current [A]

`iclamp_lower` = lower clamping current [A]

`exp_frac` = fraction of iclamp range from `iclamp_upper` and `iclamp_lower` at which exponential clamping starts to have an effect []

## Soft Voltage Clamp

### Terminals

`vin`: input terminal [V,A]  
`vout`: output terminal [V,A]  
`vgn`: gnd terminal [V,A]

### Description

`vout`- `vgn` clamped/limited to between `vclamp_upper` and `vclamp_lower` of `vin` - `vgn`.

The limiting starts working once the input voltage gets near `vclamp_lower` or `vclamp_upper`. The clamping acts exponentially to ensure smoothness.

The fraction of the range (`vclamp_lower`, `vclamp_upper`) over which the exponential clamping action occurs is `exp_frac`.

### Instance Parameters

`vclamp_upper` = upper clamping voltage [A]

`vclamp_lower` = lower clamping voltage [A]

`exp_frac` = fraction of `vclamp` range from `vclamp_upper` and `vclamp_lower` at which exponential clamping starts to have an effect []

## Self-Tuning Resistor

### Terminals

`vp`, `vn`: terminals [V,A]  
`vtune`: the voltage that is being tuned [V,A]  
`verr`: the error in `vtune` [V,A]

### Description

This element operates in four distinct phases:

1. It waits for `tsettle` seconds with the resistance between `vp` and `vn` set to `rinit`.
2. For `tdir_check` seconds, it attempts to tune the error away by increasing the resistance in proportion to the size of the error.
3. It waits for `tsettle` seconds with the resistance between `vp` and `vn` set to `rinit`.
4. For `tdir_check` seconds, it attempts to tune the error away by decreasing the resistance in proportion to the error.
5. Based on the results of (2) and (4), it selects which direction is better to tune in and tunes as best it can using integral action. For certain systems, this might lead to unstable behavior.

**Note:** Select `tsettle` to be greater than the largest system time constant. Select `rgain` so that the positive feedback is not excessive during the direction sensing phases. Select `tdir_check` so that the system has enough time to react but not so big that the resistance drifts too far from `rinit`. It is better if it can be arranged that `verr` does not change sign during tuning.

### Instance Parameters

`rmax` = maximum resistance that tuning res can have [Ohms]  
`rmin` = minimum resistance that tuning res can have [Ohms]  
`rinit` = initial resistance [Ohms]  
`rgain` = gain of integral tuning action [Ohms/(Vs)]

## Cadence Verilog-A Language Reference

### Sample Model Library

---

`vtune_set` = value that `vtune` must be tuned to [V]

`tsettle` = amount of time to wait before tuning begins [s]

`tdir_check` = amount of time to spend checking each tuning direction [s]

## Untrimmed Capacitor

### Terminals

`vp, vn:` terminals [V,A]

### Description

Each instance has a randomly generated value of capacitance, which is calculated at initialization. The distribution of these random values is gaussian (that is, normal) with a `c_mean` and a standard deviation of `c_std`.

Two seeds are needed to generate the gaussian distribution.

### Instance Parameters

`c_mean` = mean capacitance [Ohms]

`c_dev` = standard deviation of capacitance [Ohms]

`seed1` = first seed value for randomly generating capacitance values []

`seed2` = second seed value for randomly generating capacitance values []

`show_val` = option to print the value of capacitance to stdout

## Untrimmed Inductor

### Terminals

`vp, vn:` terminals [V,A]

### Description

Each instance has a randomly generated value of inductance, which is calculated at initialization. The distribution of these random values is gaussian (that is, normal) with an `l_mean` and a standard deviation of `l_std`.

Two seeds are needed to generate the gaussian distribution.

### Instance Parameters

`l_mean` = mean inductance [Ohms]

`l_dev` = standard deviation of inductance [Ohms]

`seed1` = first seed value for randomly generating inductance values []

`seed2` = second seed value for randomly generating inductance values []

`show_val` = option to print the value of inductance to stdout



## Untrimmed Resistor

### Terminals

`vp, vn:` terminals [V,A]

### Description

Each instance has a randomly generated value of resistance, which is calculated at initialization. The distribution of these random values is gaussian (that is, normal) with an `r_mean` and a standard deviation of `r_std`.

Two seeds are needed to generate the gaussian distribution.

### Instance Parameters

`r_mean` = mean resistance [Ohms]

`r_dev` = standard deviation of resistance [Ohms]

`seed1` = first seed value for randomly generating resistance values []

`seed2` = second seed value for randomly generating resistance values []

`show_val` = option to print the value of resistance to stdout

## Voltage Deadband Amplifier

### Terminals

`vin_p, vin_n`: differential input voltage terminals [V,A]

`vout`: output voltage terminal [V,A]

### Description

Outputs `vleak` when differential input voltage (`vin_p-vin_n`) is between `vdead_low` and `vdead_high`. When outside the deadband, the output voltage is an amplified version of the differential input voltage plus `vleak`.

### Instance Parameters

`vdead_low` = lower range of dead band [V]

`vdead_high` = upper range of dead band [V]

`vleak` = offset voltage; only output in deadband [V]

`gain_low` = differential voltage gain in lower region []

`gain_high` = differential voltage gain in upper region []

## Voltage-Controlled Variable-Gain Amplifier

### Terminals

`vin_p, vin_n`: differential input terminals [V,A]  
`vctrl_p, vctrl_n`: differential-controlling voltage terminals [V,A]  
`vout`: [V,A]

### Description

When there is no input offset voltage, the output is  $vout = gain\_const * (vctrl\_p - vctrl\_n) * (vin\_p - vin\_n) + (vout\_high + vout\_low)/2$ .

When there is an input offset voltage, `vin_offset` is subtracted from  $(vin\_p - vin\_n)$ .

### Instance Parameters

`gain_const` = amplifier gain when  $(vctrl\_p - vctrl\_n) = 1$  volt []  
`vout_high` = upper output limit [V]  
`vout_low` = lower output limit [V]  
`vin_offset` = input offset [V]

## Basic Components

### Resistor

#### Terminals

`vp`, `vn`: terminals (V,A)

#### Instance Parameters

`r` = resistance (Ohms)

## Capacitor

### Terminals

`vp`, `vn`: terminals (V,A)

### Instance Parameters

`c` = capacitance (F)

## **Inductor**

### **Terminals**

`vp`, `vn`: terminals (V,A)

### **Instance Parameters**

`l` = inductance (H)

## **Voltage-Controlled Voltage Source**

### **Terminals**

vout\_p, vout\_n:      controlled voltage terminals [V,A]

vin\_p, vin\_n:        controlling voltage terminals [V,A]

### **Instance Parameters**

gain = voltage gain []

## **Current-Controlled Voltage Source**

### **Terminals**

`vout_p, vout_n:` controlled voltage terminals [V,A]

`iin_p, iin_n:` controlling current terminals [V,A]

### **Instance Parameters**

`rm` = resistance multiplier (V to I gain) [Ohms]



## **Voltage-Controlled Current Source**

### **Terminals**

`iout_p, iout_n:` controlled current source terminals [V,A]

`vin_p, vin_n:` controlling voltage terminals [V,A]

### **Instance Parameters**

`gm` = conductance multiplier (V to I gain) [Mhos]

## **Current-Controlled Current Source**

### **Terminals**

`iout_p, iout_n:` controlled current terminals [V,A]

`iin_p, iin_n:` controlling current terminals [V,A]

### **Instance Parameters**

`gain = current gain []`

## Switch

### Terminals

`vp`, `vn`: output terminals [V,A]

`vctrlp`, `vctrln`: control terminals [V,A]

### Description

If  $(vctrlp - vctrln > vth)$ , the branch between `vp` and `vn` is shorted. Otherwise, the branch between `vp` and `vn` is opened.

### Instance Parameters

`vth` = threshold voltage [V]

## Control Components

### Error Calculation Block

#### Terminals

`sigset`: setpoint signal (val, flow)

`sigact`: actual value signal (val, flow)

`sigerr`: error: difference between signals (val, flow)

#### Description

`sigerr = sigset - sigact`

**Note:** Defining larger values of `abstol` and `huge` for the quantities associated with `signin` and `sigout` can help overcome convergence and clipping problems.

#### Instance Parameters

`tdel, trise, tfall = {usual}`

## Lag Compensator

### Terminals

sigin: (val, flow)

sigout: (val, flow)

### Description

$$TF = gain \times alpha \times \frac{1 + tau \times S}{1 + alpha \times tau \times S}$$

**Note:** Defining larger values of `abstol` and `huge` for the quantities associated with `sigin` and `sigout` can help overcome convergence and clipping problems.

### Instance Parameters

`gain` = compensator gain []

`tau` = compensator zero at  $-(1/tau)$  [s]

`alpha` = compensator pole at  $-(1/(alpha*tau))$ ;  $alpha > 1$  []

## Lead Compensator

### Terminals

sigin: (val, flow)

sigout: (val, flow)

### Description

$$TF = gain \times alpha \times \frac{1 + tau \times S}{1 + alpha \times tau \times S}$$

**Note:** Defining larger values of `abstol` and `huge` for the quantities associated with `sigin` and `sigout` can help overcome convergence and clipping problems.

### Instance Parameters

`gain` = compensator gain []

`tau` = compensator zero at  $-(1/tau)$  [s]

`alpha` = compensator pole at  $-(1/(alpha*tau))$ ; `alpha` < 1 []

## Lead-Lag Compensator

### Terminals

signin: (val, flow)

sigout: (val, flow)

### Description

$TF =$

$$gain \times alpha1 \times \frac{1 + tau1 \times S}{1 + alpha1 \times tau1 \times S} \times alpha2 \times \frac{1 + tau2 \times S}{1 + alpha2 \times tau2 \times S}$$

Defining larger values of `abstol` and `huge` for the quantities associated with `signin` and `sigout` can help overcome convergence and clipping problems.

### Instance Parameters

`gain` = compensator gain []

`tau1` = compensator zero at  $-(1/tau1)$  [s]

`alpha1` = compensator pole at  $-(1/(alpha*tau1))$ ; `alpha1` > 1 []

`tau2` = compensator zero at  $-(1/tau2)$  [s]

`alpha2` = compensator pole at  $-(1/(alpha*tau2))$ ; `alpha2` < 1 []

## Proportional Controller

### Terminals

`signin:` (val, flow)

`sigout:` (val, flow)

### Description

`sigout = kp*signin`

**Note:** Defining larger values of `abstol` and `huge` for the quantities associated with `signin` and `sigout` can help overcome convergence and clipping problems.

### Instance Parameters

`kp` = proportional gain []



## Proportional Derivative Controller

### Terminals

signin: (val, flow)

sigout: (val, flow)

### Description

$\text{sigout} = k_p \cdot \text{signin} + k_d \cdot \text{dot}(\text{signin})$

**Note:** Defining larger values of `abstol` and `huge` for the quantities associated with `signin` and `sigout` can help overcome convergence and clipping problems.

### Instance Parameters

`kp` = proportional gain []

`kd` = differential gain []

## Proportional Integral Controller

### Terminals

sigin: (val, flow)

sigout: (val, flow)

### Description

This model is a proportional, integral, and derivative controller.

$$\text{sigout} = k_p * \text{sigin} + k_i * \text{integ}(\text{sigin}) + k_d * \text{dot}(\text{sigin})$$

**Note:** Defining larger values of `abstol` and `huge` for the quantities associated with `sigin` and `sigout` can help overcome convergence and clipping problems.

### Instance Parameters

`kp` = proportional gain []

`ki` = integral gain []

## Proportional Integral Derivative Controller

### Terminals

signin: (val, flow)

sigout: (val, flow)

### Description

$\text{sigout} = k_p * \text{signin} + k_i * \text{integ}(\text{signin}) + k_d * \text{dot}(\text{signin})$

**Note:** Defining larger values of `abstol` and `huge` for the quantities associated with `signin` and `sigout` can help overcome convergence and clipping problems.

### Instance Parameters

`kp` = proportional gain []

`ki` = integral gain []

`kd` = differential gain []

## Logic Components

### AND Gate

#### Terminals

vin1, vin2: [V,A]

vout: [V,A]

#### Instance Parameters

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## **NAND Gate**

### **Terminals**

vin1, vin2:        [V,A]

vout:             [V,A]

### **Instance Parameters**

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for high [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## **OR Gate**

### **Terminals**

vin1, vin2:        [V,A]

vout:              [V,A]

### **Instance Parameters**

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for high [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## **NOT Gate**

### **Terminals**

vin:           [V,A]

vout:           [V,A]

### **Instance Parameters**

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for high [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## **NOR Gate**

### **Terminals**

vin1, vin2:        [V,A]

vout:              [V,A]

### **Instance Parameters**

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for high [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]



## **XOR Gate**

### **Terminals**

vin1, vin2:     [V,A]

vout:           [V,A]

### **Instance Parameters**

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for high [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## **XNOR Gate**

### **Terminals**

vin1, vin2:     [V,A]

vout:           [V,A]

### **Instance Parameters**

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for high [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## **D-Type Flip-Flop**

### **Terminals**

vin\_d: [V,A]

vclk: [V,A]

out\_q, vout\_qbar: [V,A]

### **Description**

Triggered on the rising edge.

### **Instance Parameters**

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

vtrans = voltages above this at input are considered high [V]

vtrans\_clk = transition voltage of clock [V]

t<sub>del</sub>, t<sub>rise</sub>, t<sub>fall</sub> = {usual} [s]

## Clocked JK Flip-Flop

### Terminals

vin\_j: [V,A]

vin\_k: [V,A]

vclk: [V,A]

vout\_q: [V,A]

vout\_qbar: [V,A]

### Description

Triggered on the rising edge.

### Logic Table

J	K	Q	Q'
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

### Instance Parameters

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

vtrans = voltages above this at input are considered high [V]

## Cadence Verilog-A Language Reference

### Sample Model Library

---

`tdel, trise, tfall = {usual} [s]`

## JK-Type Flip-Flop

### Terminals

vin\_j, vin\_k: inputs

vout\_q, vout\_qbar: outputs

### Description

Triggered on the rising edge.

### Logic Table

J	K	Q	Q(t+e)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

### Instance Parameters

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## **Level Shifter**

### **Terminals**

sigin: (val, flow)

sigout: (val, flow)

### **Description**

sigout = sigin added to sigshift.

### **Instance Parameters**

sigshift = level shift (val)

## RS-Type Flip-Flop

### Terminals

vin\_s: [V,A]

vin\_r: [V,A]

vout\_q, vout\_qbar: [V,A]

### Logic Table

S(t)	R(t)	Q(t)	Q(t+e)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	X
1	1	1	X

### Instance Parameters

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]



## Trigger-Type (Toggle-Type) Flip-Flop

### Terminals

vtrig:     trigger [V,A]

vout\_q, vout\_qbar:     outputs [V,A]

### Description

Triggered on the rising edge.

### Logic Table

T	Q	Q(t+e)
0	0	0
0	1	1
1	0	1
1	1	0

### Instance Parameters

initial\_state = the initial state/output of the flip-flop []

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## Half Adder

### Terminals

vin1, vin2: bits to be added [V,A]

vout\_sum: vout\_sum out [V,A]

vout\_carry: carry out [V,A]

### Instance Parameters

vlogic\_high = logic high value [V]

vlogic\_low = logic low value [V]

vtrans = threshold for inputs to be high [V]

tdel, trise, tfall = {usual} [s]

## Full Adder

### Terminals

vin1, vin2: bits to be added [V,A]

vin\_carry: carry in [V,A]

vout\_sum: sum out [V,A]

vout\_carry: carry out [V,A]

### Instance Parameters

vlogic\_high = logic high value [V]

vlogic\_low = logic low value [V]

vtrans = threshold for inputs to be high [V]

tdel, trise, tfall = {usual} [s]

## Half Subtractor

### Terminals

vin1, vin2:        inputs [V,A]  
vout\_diff:        difference out [V,A]  
vout\_borrow:     borrow out [V,A]

### Formula

vin1 - vin2 = vout\_diff and borrow

### Truth Table

in1	in2	diff	borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

### Instance Parameters

vlogic\_high = logic high value [V]  
vlogic\_low = logic low value [V]  
vtrans = threshold for inputs to be high [V]  
tdel, trise, tfall = {usual} [s]

## Full Subtractor

### Terminals

vin1, vin2:        inputs [V,A]  
vin\_borrow:       borrow in [V,A]  
vout\_diff:        difference out [V,A]  
vout\_borrow:     borrow out [V,A]

### Truth Table

in1	in2	bin	bout	doff
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

### Instance Parameters

vlogic\_high = logic high value [V]  
vlogic\_low = logic low value [V]  
vtrans = threshold for inputs to be high [V]  
tdel, trise, tfall = {usual} [s]

## **Parallel Register, 8-Bit**

### **Terminals**

`vin_d0..vin_d7:`      input data lines [V,A]  
`vout_d0..vout_d7:`    output data lines [V,A]  
`venable:`            enable line [V,A]

### **Description**

Input occurs on the rising edge of `venable`.

### **Instance Parameters**

`vlogic_high` = output voltage for high [V]  
`vlogic_low` = output voltage for low [V]  
`vtrans` = voltages above this at input are considered high [V]  
`tdel, trise, tfall` = {usual} [s]

## **Serial Register, 8-Bit**

### **Terminals**

`vin_d`: input data lines [V,A]  
`vout_d`: output data lines [V,A]  
`vclk`: enable line [V,A]

### **Description**

Input occurs on the rising edge of `vclk`.

### **Instance Parameters**

`vlogic_high` = output voltage for high [V]  
`vlogic_low` = output voltage for low [V]  
`vtrans` = voltages above this at input are considered high [V]  
`tdel, trise, tfall` = {usual} [s]

## Electromagnetic Components

### DC Motor

#### Terminals

`vp`: positive terminal [V,A]

`vn`: negative terminal [V,A]

`pos_shaft`: motor shaft [rad, Nm]

#### Description

This is a model of a DC motor driving a shaft.

#### Instance Parameters

`km` = motor constant [Vs/rad]

`kf` = flux constant [Nm/A]

`j` = inertia factor [Nms<sup>2</sup>/rad]

`d` = drag (friction) [Nms/rad]

`rm` = motor resistance [Ohms]

`lm` = motor inductance [H]



## Electromagnetic Relay

### Terminals

<code>vopen:</code>	normally opened terminal [V,A]
<code>vcomm:</code>	common terminal [V,A]
<code>vclosed:</code>	normally closed terminal [V,A]
<code>vctrl_n:</code>	negative control signal [V,A]
<code>vctrl_p:</code>	positive control signal [V,A]

### Description

This is a model of a voltage-controlled single-pole, double-throw switch. When the voltage differential between `vctrl_p` and `vctrl_n` exceeds `vtrig`, the normally open branch is shorted (closed). Otherwise, the normally open branch stays open. If the open branch is already closed and the voltage differential between `vctrl_p` and `vctrl_n` falls below `vrelease`, the normally open branch is opened.

### Instance Parameters

<code>vtrig</code>	= input value to close relay [V]
<code>vrelease</code>	= input value to open relay [V]

## Three-Phase Motor

### Terminals

vp1, vn1: phase 1 terminals [V,A]  
vp2, vn2: phase 2 terminals [V,A]  
vp3, vn3: phase 3 terminals [V,A]  
pos: position of shaft [rad, Nm]  
shaft: speed of shaft [rad/s, Nm]  
com: rotational reference point [rad/s, Nm]

### Instance Parameters

km = motor constant [Vs/rad]  
kf = flux constant [Nm/A]  
j = inertia factor [Nms<sup>2</sup>/rad]  
d = drag (friction) [Nms/rad]  
rm = motor resistance [Ohms]  
lm = motor inductance [H]

## Functional Blocks

### Amplifier

#### Terminals

`sigin:` input (val, flow)

`sigout:` output (val, flow)

#### Instance Parameters

`gain` = gain between input and output []

`sigin_offset` = subtracted from `sigin` before amplification (val)

## Comparator

### Terminals

`signin:` (val, flow)

`sigref:` reference to which `signin` is compared (val, flow)

`sigout:` comparator output (val, flow)

### Description

Compares (`signin`-`signin_offset`) to `sigref`—the output is related to their difference by a tanh relationship.

If the difference  $\gg \text{sigref}$ , `sigout` is `sigout_high`.

If the difference = `sigref`, `sigout` is  $(\text{sigout\_high} + \text{sigout\_low})/2$ .

If the difference  $\ll \text{sigref}$ , `sigout` is `sigout_low`.

Intermediate points are fitting to a tanh scaled by `comp_slope`.

### Instance Parameters

`sigout_high` = maximum output of the comparator (val)

`sigout_low` = minimum output of the comparator (val)

`signin_offset` = subtracted from `signin` before comparison to `sigref` (val)

`comp_slope` = determines the sensitivity of the comparator []

## Controlled Integrator

### Terminals

sigin: (val, flow)

sigout: (val, flow)

sigctrl: (val, flow)

### Description

Integration occurs while `sigctrl` is above `sigctrl_trans`.

### Instance Parameters

`sigout0` = initial `sigout` value (val)

`gain` = gain []

`sigctrl_trans` = if `sigctrl` is above this, integration occurs (val)

## Deadband

### Terminals

signin:     input (val, flow)

sigout:     output (val, flow)

### Description

Deadband region is when `signin` is between `signin_dead_high` and `signin_dead_low`. `sigout` is zero in the deadband region. Above the deadband, the output is `signin - signin_dead_high`. Below the deadband, the output is `signin - signin_dead_low`.

### Instance Parameters

`signin_dead_high` = upper deadband limit (val)

`signin_dead_low` = lower deadband limit (val)

## Deadband Differential Amplifier

### Terminals

`sigin_p, sigin_n`: differential input terminals (val, flow)

`sigout`: output terminal (val, flow)

### Description

Outputs `sigout_leak` when differential input (`sigin_p-sigin_n`) is between `sigin_dead_low` and `sigin_dead_high`. When outside the deadband, the output is an amplified version of the differential input plus `sigout_leak`.

### Instance Parameters

`sigin_dead_low` = lower range of dead band (val)

`sigin_dead_high` = upper range of dead band (val)

`sigout_leak` = offset signal; only output in deadband (val)

`gain_low` = differential gain in lower region []

`gain_high` = differential gain in upper region []

## Differential Amplifier (Opamp)

### Terminals

`sigin_p, sigin_n:` (val, flow)

`sigout:` (val, flow)

### Description

`sig_out` is `gain` times the adjusted input differential signal. The adjusted input differential signal is the differential input minus `sigin_offset`.

### Instance Parameters

`gain` = amplifier differential gain (val)

`sigin_offset` = input offset (val)



## Differential Signal Driver

### Terminals

`sigin_p, sigin_n:` differential input signals (val, flow)

`sigout_p, sigout_n:` differential output signals (val, flow)

`sigref:` differential outputs are with reference to this node  
(val, flow)

### Description

Amplifies its differential pair of input by an amount `gain`, producing a differential pair of output signals. The output differential signals appear symmetrically about `sigref`.

### Instance Parameters

`gain = diffdriver gain []`

## **Differentiator**

### **Terminals**

sigin: (val, flow)

sigout: (val, flow)

### **Instance Parameters**

gain = []

## **Flow-to-Value Converter**

### **Terminals**

signin\_p, signin\_n: [V,A]

sigout\_p, sigout\_n: [V,A]

### **Description**

$\text{val}(\text{sigout\_p}, \text{sigout\_n}) = \text{flow}(\text{signin\_p}, \text{signin\_n})$

### **Instance Parameters**

gain = flow to val gain

## Rectangular Hysteresis

### Terminals

sigin: (flow, val)

sigout: (flow, val)

### Instance Parameters

hyst\_state\_init = the initial output []

sigout\_high = maximum input/output (val)

sigout\_low = minimum input/output (val)

sigtrig\_low = the sigin value that will cause sigout to go low when sigout is high (val)

sigtrig\_high = the sigin value that will cause sigout to go high when sigout is low (val)

tdel, trise, tfall = {usual} [s]

## **Integrator**

### **Terminals**

sigin: (val, flow)

sigout: (val, flow)

### **Instance Parameters**

sigout0 = initial sigout value (val)

gain = []

## **Level Shifter**

### **Terminals**

sigin: (val, flow)

sigout: (val, flow)

### **Description**

sigout = sigin added to sigshift.

### **Instance Parameters**

sigshift = level shift (val)

## Limiting Differential Amplifier

### Terminals

`sigin_p, sigin_n:` (val, flow)

`sigout:` (val, flow)

### Description

Has limited output swing. `sigout` is `gain` times the adjusted differential input signal about  $(\text{sigout\_high} + \text{sigout\_low})/2$ . The adjusted differential input signal is the differential input signal minus `sigin_offset`.

### Instance Parameters

`sigout_high` = upper amplifier output limit (val)

`sigout_low` = lower amplifier output limit (val)

`gain` = amplifier gain within the limits []

`sigin_offset` = input offset (val)

## Logarithmic Amplifier

### Terminals

signin: (val, flow)

sigout: (val, flow)

### Description

sigout is gain times the natural log of the absolute value of the adjusted input. The adjusted input is signin minus signin\_offset unless the absolute value of the this is less than min\_signin. In this case, min\_signin is used as the adjusted input.

### Instance Parameters

min\_signin = absolute value of minimum acceptable signin (val)

gain = (val)

signin\_offset = input offset (val)



## Multiplexer

### Terminals

signin1, signin2, signin3: signals to be multiplexed (val, flow)

cntrlp, cntrlm: differential-controlling signal (val, flow)

sigout: (val, flow)

### Description

If the differential-controlling signal is below `sigth_high`, `sigout` is `signin1`. If the differential-controlling signal is above `sigth_low`, `sigout` is `signin3`. In between these two thresholds, `sigout = signin2`.

### Instance Parameters

`sigth_high` = high threshold value (val)

`sigth_low` = low threshold value (val)

## Quantizer

### Terminals

sigin: (val, flow)

sigout: (val, flow)

### Description

This model quantizes input with unity gain.

### Instance Parameters

nlevel = number of levels to quantize to []

round = if yes, go to nearest q-level, otherwise go to nearest q-level below []

sigout\_high = maximum input/output (val)

sigout\_low = minimum input/output (val)

tdel, trise, tfall = {usual} [s]

## Repeater

### Terminals

signin: (val, flow)

sigout: (val, flow)

### Description

From 0 to `period`, `sigout` = `signin`. After this, `sigout` is a periodic repetition of what `signin` was between 0 and `period`.

### Instance Parameters

`period` = period of repeated waveform (val)

## **Saturating Integrator**

### **Terminals**

sigin: (val, flow)

sigout: (val, flow)

### **Description**

The output is the limited integral of the input. The limits are `sigout_max`, `sigin_min`. `sigout0` must lie between `sigout_max` and `sigin_min`.

### **Instance Parameters**

`sigout0` = initial sigout value (val)

`gain` = []

`sigout_max` = maximum signal out (val)

`sigout_min` = minimum signal out (val)

## Swept Sinusoidal Source

### Terminals

sigout\_p, sigout\_n:     output (val, flow)

### Description

The instantaneous frequency of the output is  $\text{sweep\_rate} * \text{time}$  plus  $\text{start\_freq}$ .

### Instance Parameters

$\text{start\_freq}$  = start frequency [Hz]

$\text{sweep\_rate}$  = rate of increase in frequency [Hz/s]

$\text{amp}$  = amplitude of output sinusoid (val)

$\text{points\_per\_cycle}$  = number of points in a cycle of the output []

## Three-Phase Source

### Terminals

`vouta`: A-phase terminal [V,A]

`voutb`: B-phase terminal [V,A]

`voutc`: C-phase terminal [V,A]

`vout_star`: star terminal [V,A]

### Instance Parameters

`amp` = phase-to-phase voltage amplitude [V]

`freq` = output frequency [Hz]

## **Value-to-Flow Converter**

### **Terminals**

sigin\_p, sigin\_n: [V,A]

sigout\_p, sigout\_n: [V,A]

### **Description**

$\text{flow}(\text{sigout\_p}, \text{sigout\_n}) = \text{val}(\text{sigin\_p}, \text{sigin\_n})$

### **Instance Parameters**

gain = value-to-flow gain []

## Variable Frequency Sinusoidal Source

### Terminals

`signin`: frequency-controlling signal (val, flow)

`sigout`: (val, flow)

### Description

Outputs a variable frequency sinusoidal signal. Its instantaneous frequency is  $(\text{center\_freq} + \text{freq\_gain} * \text{signin})$  [Hz]

### Instance Parameters

`amp` = amplitude of the output signal (val)

`center_freq` = center frequency of oscillation frequency when `signin = 0` [Hz]

`freq_gain` = oscillator conversion gain (Hz/val)



## Variable-Gain Differential Amplifier

### Terminals

`sigin_p, sigin_n`: differential input terminals (val, flow)

`sigctrl_p, sigctrl_n`: differential-controlling terminals (val, flow)

`sigout`: (val, flow)

### Description

`sigout` is the product of `gain_const`,  $(\text{sigctrl\_p} - \text{sigctrl\_n})$ , and the adjusted input differential signal added to  $(\text{sigout\_high} + \text{sigout\_low})/2$ . The adjusted input differential signal is the input differential signal minus `sigin_offset`.

### Instance Parameters

`gain_const` = amplifier gain when  $(\text{sigctrl\_p} - \text{sigctrl\_n}) = 1$  unit []

`sigout_high` = upper output limit (val)

`sigout_low` = lower output limit (val)

`sigin_offset` = input offset (val)

## Magnetic Components

### Magnetic Core

#### Terminals

mp: positive MMF terminal [A, Wb]

mn: negative MMF terminal [A, Wb]

#### Description

This is a Jiles/Atherton magnetic core model.

#### Instance Parameters

len = effective magnetic length of core [m]

area = magnetic cross-section area of core [m<sup>2</sup>]

ms = saturation magnetization

gamma = shaping coefficient

k = bulk coupling coefficient

alpha = interdomain coupling coefficient

c = coefficient for reversible magnetization

## **Magnetic Gap**

### **Terminals**

mp: positive MMF terminal [A, Wb]

mn: negative MMF terminal [A, Wb]

### **Description**

This is a Jiles/Atherton magnetic gap model.

This model is analogous to a linear resistor in an electrical system.

### **Instance Parameters**

len = effective magnetic length of gap [m]

area = magnetic cross-section area of gap [m<sup>2</sup>]

## **Magnetic Winding**

### **Terminals**

- vp: positive voltage terminal [V,A]
- vn: negative voltage terminal [V,A]
- mp: positive MMF terminal [A, Wb]
- mn: negative MMF terminal [A, Wb]

### **Description**

This is a Jiles/Atherton winding model.

### **Instance Parameters**

- num\_turns = number of turns []
- rturn = winding resistance per turn [Ohms]

## Two-Phase Transformer

### Terminals

vp\_1, vn\_1: [V,A]

vp\_2, vn\_2: [V,A]

### Description

This is structural transformer model implemented using Jiles/Atherton core and winding primitives

### Instance Parameters

turns1 = number of turns in the first winding []

turns2 = number of turns in the second winding []

rwinding1 = resistance per turn of first winding [Ohms]

rwinding2 = resistance per turn of second winding [Ohms]

len = length of the transformer core [m]

area = area of the transformer core [m<sup>2</sup>]

ms = saturation magnetization

gamma = shaping coefficient

k = bulk coupling coefficient

alpha = interdomain coupling coefficient

c = coefficient for reversible magnetization

## Mathematical Components

### Absolute Value

#### Terminals

signin: (val, flow)

sigout: (val, flow)

#### Description

sigout is the absolute value of signin.

#### Instance Parameters

None.

## **Adder**

### **Terminals**

sigin1, sigin2: (val, flow)

sigout: (val, flow)

### **Description**

This model adds two node values.

### **Instance Parameters**

k1 = gain of sigin1 []

k2 = gain of sigin2 []

## **Adder, 4 Numbers**

### **Terminals**

signin1, signin2, signin3, signin4:     (val, flow)

sigout:                     (val, flow)

### **Description**

$\text{sigout} = \text{gain1} * \text{signin1} + \text{gain2} * \text{signin2} + \text{gain3} * \text{signin3} + \text{gain4} * \text{signin4}$

### **Instance Parameters**

gain1 = gain for signin1 []

gain2 = gain for signin2 []

gain3 = gain for signin3 []

gain4 = gain for signin4 []



## **Cube**

### **Terminals**

signin: (val, flow)

sigout: (val, flow)

### **Description**

sigout is the cube of the signin.

### **Instance Parameters**

None.

## **Cubic Root**

### **Terminals**

`signin:` (val, flow)

`sigout:` (val, flow)

### **Description**

`sigout` is the cubic root of `signin`.

### **Instance Parameters**

`epsilon` = small number added to `signin` to ensure not getting `pow(0,0.3333.)`, because `pow()` is implemented using `logs (val)`

## Divider

### Terminals

signumer:     numerator (val, flow)

sigdenom:     denominator (val, flow)

sigout:       (val, flow)

### Description

sigout is gain multiplied by signumer divided by sigdenom unless the absolute value of sigdenom is less than min\_sigdenom. In that case, signumer is divided by min\_sigdenom instead and multiplied by the sign of the sigdenom.

### Instance Parameters

gain = divider gain []

min\_sigdenom = minimum denominator (val)

## Exponential Function

### Terminals

`signin:` (val, flow)

`sigout:` (val, flow)

### Description

`sigout` is an exponential function of `signin`. However, if `signin` is greater than `max_signin`, `signin` is taken to be `max_signin`. This is necessary because the exponential function explodes very quickly.

### Instance Parameters

`max_signin` = maximum value of `signin` accepted (val)

## **Multiplier**

### **Terminals**

sigin1, sigin2:     inputs (val, flow)

sigout:            terminals (val, flow)

### **Description**

`sigout = gain * sigin1 * sigin2`

### **Instance Parameters**

`gain = gain of multiplier []`

## Natural Log Function

### Terminals

signin: (val, flow)

sigout: (val, flow)

### Description

sigout is the natural log of signin, providing  $\text{signin} > \text{min\_signin}$ . If  $\text{signin}$  is between 0 and  $\text{min\_signin}$ , sigout is the log of  $\text{min\_signin}$ . If  $\text{signin}$  is less than 0, an error is reported.

### Instance Parameters

$\text{min\_signin}$  = minimum value of  $\text{signin}$  (val)

## Polynomial

### Terminals

sigin: (val, flow)

sigout: (val, flow)

### Description

This is a model of a third-order polynomial function.

$$\text{sigout} = p3 * \text{sigin}^3 + p2 * \text{sigin}^2 + p1 * \text{sigin} + p0$$

### Instance Parameters

p3 = cubic coefficient []

p2 = square coefficient []

p1 = linear coefficient []

p0 = constant coefficient []

## Power Function

### Terminals

`signin:` (val, flow)

`sigout:` (val, flow)

### Description

`sigout` is `signin` to the power of `exponent`.

### Instance Parameters

`exponent` = what `signin` is raised by []

`epsilon` = small number added to `signin` to ensure not getting `pow(0,0.3333.)`, because `pow()` is implemented using logs (val)



## **Reciprocal**

### **Terminals**

sigin: (val, flow)

sigout: (val, flow)

### **Description**

sigout is gain/denom

### **Instance Parameters**

gain = gain (val)

min\_sigdenom = minimum denominator (val)

## **Signed Number**

### **Terminals**

signin: (val, flow)

sigout: (val, flow)

### **Description**

This is a model of the sign of the input.

sigout is +1 if signin  $\geq$  0; otherwise, sigout is -1.

### **Instance Parameters**

None.

## **Square**

### **Terminals**

signin:   input

sigout:   output

### **Description**

sigout is the square of the signin.

### **Instance Parameters**

None.

## Square Root

### Terminals

`signin:` (val, flow)

`sigout:` (val, flow)

### Description

`sigout` is the square root of `signin`.

### Instance Parameters

None.

## **Subtractor**

### **Terminals**

`sigin_p`:     input subtracted from (val, flow)  
`sigin_n`:     input that is subtracted (val, flow)  
`sigout`:     (val, flow)

### **Instance Parameters**

None.

## **Subtractor, 4 Numbers**

### **Terminals**

signin1, signin2, signin3, signin4: (val, flow)

sigout: (val, flow)

### **Description**

$\text{sigout} = \text{gain1} * \text{signin1} - \text{gain2} * \text{signin2} - \text{gain3} * \text{signin3} - \text{gain4} * \text{signin4}$

### **Instance Parameters**

gain1 = gain for signin1

gain2 = gain for signin2

gain3 = gain for signin3

gain4 = gain for signin4

## Measure Components

### ADC, 8-Bit Differential Nonlinearity Measurement

#### Terminals

`vd0..vd7`: data lines from ADC [V,A]  
`vout`: voltage sent from conversion to ADC [V,A]  
`vc1k`: clocking signal for the ADC [V,A]

#### Description

Measures an 8-bit analog-to-digital converter's (ADC's) differential nonlinearity measurement (DNL) using a histogram method. `vout` is sequentially set to 4,096 equally spaced voltages between `vstart` and `vend`. At each different value of `vout`, a clock pulse is generated causing the ADC to convert this `vout` value. The resultant code of each conversion is stored.

When all the conversions have been done, the DNL is calculated from the recorded data.

If `log_to_file` is `yes`, the DNL (differential nonlinearity) is recorded and written to `filename`.

#### Instance Parameters

`vlogic_high` = [V]

`vlogic_low` = [V]

`tsettle` = time to allow for settling after the data lines are changed before `vd0-7` are recorded [s]—also the period of the ADC conversion clock.

`vstart` = voltage at which to start conversion sweep []

`vend` = voltage at which to end conversion sweep []

`log_to_file` = whether to log the results to a file; `yes` or `no` []

`filename` = the name of the file in which the results are logged []

## ADC, 8-Bit Integral Nonlinearity Measurement

### Terminals

`vd0..vd7`: data lines from ADC [V,A]  
`vout`: voltage sent from conversion to ADC [V,A]  
`vclk`: clocking signal for the ADC [V,A]

### Description

Measures an 8-bit ADC's INL using a histogram method. `vout` is sequentially set to 4,096 equally spaced voltages between `vstart` and `vend`. At each different value of `vout`, a clock pulse is generated causing the ADC to convert this `vout` value. The resultant code of each conversion is stored.

When all the conversions have been done, the INL is calculated from the recorded data.

If `log_to_file` is `yes`, the INL (integral nonlinearity) is recorded and written to `filename`.

### Instance Parameters

`vlogic_high` = [V]  
`vlogic_low` = [V]  
`tsettle` = time to allow for settling after the data lines are changed before `vd0-7` are recorded [s]—also the period of the ADC conversion clock.  
`vstart` = voltage at which to start conversion sweep []  
`vend` = voltage at which to end conversion sweep []  
`log_to_file` = whether to log the results to a file; `yes` or `no` []  
`filename` = the name of the file in which the results are logged []



## Ammeter (Current Meter)

### Terminals

`vp`, `vn`: terminals [V,A]

`vout`: measured current converted to a voltage [V,A]

### Description

Measures the current between two of its nodes. It has two modes: rms (root-mean-squared) and absolute.

The measurement is passed through a first-order filter with bandwidth `bw` before being written to a file and appearing at `vout`. This is useful when doing rms measurements. If `bw` is set to zero, no filtering is done.

### Instance Parameters

`mtype` = type of current measurement; absolute or rms []

`bw` = bw of output filter (a first-order filter) [Hz]

`log_to_file` = whether to log the results to a file; yes or no []

`filename` = the name of the file in which the results are logged []

## DAC, 8-Bit Differential Nonlinearity Measurement

### Terminals

`vin`: terminal for monitoring DAC output voltages [V,A]

`vd0..vd7`: data lines for DAC [V,A]

### Description

Sweeps through all the 256 codes and records the digital-to-analog converter (DAC) output voltage and writes the maximum DNL found to the output.

If `log_to_file` is `yes`, the DNL (differential nonlinearity) is recorded and written to `filename`.

### Instance Parameters

`vlogic_high` = [V]

`vlogic_low` = [V]

`tsettle` = time to allow for settling after the data lines are changed before `vin` is recorded [s]

`log_to_file` = whether to log the results to a file; `yes` or `no` []

`filename` = the name of the file in which the results are logged []

## DAC, 8-Bit Integral Nonlinearity Measurement

### Terminals

`vin`: terminal for monitoring DAC output voltages [V,A]

`vd0..vd7`: data lines for DAC [V,A]

### Description

Sweeps through all the 256 codes and records the DAC output voltage and writes the maximum INL found to the output.

If `log_to_file` is `yes`, the INL (integral nonlinearity) is recorded and written to *filename*.

### Instance Parameters

`vlogic_high` = [V]

`vlogic_low` = [V]

`tsettle` = time to allow for settling after the data lines are changed before `vin` is recorded [s]

`log_to_file` = whether to log the results to a file; `yes` or `no` []

`filename` = the name of the file in which the results are logged []

## Delta Probe

### Terminals

`start_pos, start_neg`: signal that controls start of measurement []

`stop_pos, stop_neg`: signal that controls end of measurement []

### Description

This probe measures argument delta between the occurrence of the starting and stopping events. It can also be used to find when the start and stop signals cross the specified reference values (by default `start_count` and `stop_count` are set to 1).

### Instance Parameters

`start_td, stop_td` = signal delays [s]

`start_val, stop_val` = signal value that starts/end measurement []

`start_count, stop_count` = number of signal values that starts/end measurement

`start_mode` = one of the starting/stopping modes []

arg—argument value (simulation time)

rise—crossing of the signal value on rise

fall—crossing of the signal value on fall

crossing—any crossing of the signal value

`stop_mode` = one of the starting/stopping modes []

arg—argument value (simulation time)

rise—crossing of the signal value on rise

fall—crossing of the signal value on fall

crossing—any crossing of the signal value

## Find Event Probe

### Terminals

`out_pos, out_neg:`            signal to measure []  
`start_pos, start_neg:`        signal that controls start of measurement []  
`ref_pos, ref_neg:`            differential reference signal

### Description

This model is of a signal statistics probe. This probe measures the output signal at the occurrence of the event:

- If `arg_val` is given, measure at this value.
- If `start_ref_val` is given, measure the output signal when the start signal crosses this value.
- If `start_ref_val` is not given, measure the output signal when it is equal to the reference signal.

### Instance Parameters

`start` = argument value that starts measurements  
`stop` = argument value that stops measurements  
`start_td` = signal delays [s]  
`start_val` = signal value that starts/ends measurement []  
`start_count` = number of signal values that starts/ends measurement  
`start_mode` = one of the starting/stopping modes []  
    `arg`—argument value (simulation time)  
    `rise`—crossing of the signal value on rise  
    `fall`—crossing of the signal value on fall  
    `crossing`—any crossing of the signal value

## Cadence Verilog-A Language Reference

### Sample Model Library

---

`start_ref_val` = start signal reference value []

`arg_val` = argument value that controls when to measure signals []

1. If `arg_val` is given, measure at the specified value of the simulation argument. If it is not given, measure at the occurrence of the event.
2. If `start_ref_val` is given, measure the output signal when the start signal is equal to the reference value.
3. If `start_ref_val` is not given, measure the output signal when the start signal is equal to the reference signal.

## Find Slope

### Terminals

out\_pos, out\_neg:            signal to measure []

### Description

This model is of a signal statistics probe.

This probe measures slope of a signal between `arg_val1` and `arg_val2`; if `arg_val2` is not specified, it is set to the value exceeding `arg_val1` by 0.1%.

### Instance Parameters

`arg_val1` = first argument value []

`arg_val2` = (optional) second argument value []

## Frequency Meter

### Terminals

`vp, vn:` terminals [V,A]

`fout:` measured frequency [F,A]

### Description

Measures the frequency of the voltage across the terminals by detecting the times at which the last two zero crossings occurred. This method only works on pure AC waveforms.

### Instance Parameters

`log_to_file` = whether to log the results to a file; yes or no []

`filename` = the name of the file in which the results are logged []



## Offset Measurement

### Terminals

<code>vamp_out:</code>	output voltage of opamp being measured [V,A]
<code>vamp_p:</code>	positive terminal of opamp being measured [V,A]
<code>vamp_n:</code>	negative terminal of opamp being measured [V,A]
<code>vamp_supply_p:</code>	positive supply of opamp being measured [V,A]
<code>vamp_supply_n:</code>	negative supply of opamp being measured [V,A]

### Description

This is a model of a slew rate measurer.

The opamp terminals of the opamp under test are connected to this model. It shorts `vamp_out` to `vamp_n` and grounds `vamp_vp`. After `tsettle` seconds, the voltage read at `vamp_out` is taken to be `offset`.

The result is printed to the screen.

### Instance Parameters

<code>vsupply_p</code>	= positive supply voltage required by opamp [V]
<code>vsupply_n</code>	= negative supply voltage required by opamp [V]
<code>tsettle</code>	= time to let opamp settle before measuring the offset [s]

## Power Meter

### Terminals

- `iin`: input for current passing through the meter [V,A]
- `vp_iout`: positive voltage sensing terminal and output for current passing through the meter [V,A]
- `vn`: negative voltage sensing terminal [V,A]
- `pout`: measured impedance converted to a voltage [V]
- `va_out`: measured apparent power [W]
- `pf_out`: measured power factor []

### Description

To measure the power being dissipated in a 2-port device, this meter should be placed in the netlist so that the current flowing into the device passes between `iin` and `vp_iout` first, that `vp_iout` is connected to the positive terminal of the device, and that `vn` is connected to the negative terminal of the device.

The measured power is the average over time of the product of the voltage across and the current through the device. This average is calculated by integrating the VI product and dividing by time and passing the result through a first-order filter with bandwidth `bw`.

The apparent power is calculated by finding the rms values of the current and voltage first and filtering them with a first-order filter of bandwidth `bw`. The apparent power is the product of the voltage and current rms values.

The purpose of the filtering is to remove ripple. Cadence recommends that `bw` be set to a low value to produce accurate measurements and that at least 10 input AC cycles be allowed before the power meter is considered settled. Also allow time for the filters to settle.

This meter requires accurate integration, so it is desirable that the integration method is set to `gear2only` in the netlist.

### Instance Parameters

`tstart` = time to wait before starting measurement [s]

## Cadence Verilog-A Language Reference

### Sample Model Library

---

`bw` = bw of rms filters (a first-order filter) [Hz]

`log_to_file` = whether to log the results to a file; `yes` or `no` []

`filename` = the name of the file in which the results are logged []

## **Q (Charge) Meter**

### **Terminals**

`vp, vn`: terminals [V,A]

`qout`: measured charge [C,A]

### **Description**

Measures the charge that has flown between `vn` and `vp` between `tstart` and `tend`.

### **Instance Parameters**

`tstart` = start time [s]

`tend` = end time [s]

`log_to_file` = whether to log the results to a file; yes or no []

`filename` = the name of the file in which the results are logged []

## Sampler

### Terminal

`signin: (val, flow)`

### Description

Samples `signin` every `tsample` and writes the results to `filename` and labels the data with `label`. The time variable is recorded if `log_time` is `yes`.

### Instance Parameters

`tsample` = how often input is sampled [s]

`filename` = name of file where samples are stored []

`label` = label for signal being sampled []

`log_time` = if the time variable should be logged to a file []

## Slew Rate Measurement

### Terminals

<code>vamp_out:</code>	output voltage of the opamp being measured [V,A]
<code>vamp_p:</code>	positive terminal of the opamp being measured [V,A]
<code>vamp_n:</code>	negative terminal of the opamp being measured [V,A]
<code>vamp_spplly_p:</code>	positive supply of the opamp being measured [V,A]
<code>vamp_spplly_n:</code>	negative supply of the opamp being measured [V,A]

### Description

Monitors the input and records the times at which it equals `vstart` and `vend`. The slew is given to be `vstart - vend` divided by the time difference.

The result is printed to the screen.

### Instance Parameters

<code>vspply_p</code>	= positive supply voltage required by opamp [V]
<code>vspply_n</code>	= negative supply voltage required by opamp [V]
<code>twait</code>	= time to wait before applying pulse to opamp input [V]
<code>vstart</code>	= voltage at which to record the first measurement point [V]
<code>vend</code>	= voltage at which to record the other measurement point [V]
<code>tmin</code>	= minimum time allowed between both measurements before an error is reported [s]

## Signal Statistics Probe

### Terminals

out\_pos, out\_neg:            signal to measure []  
start\_pos, start\_neg:        signal that controls start of measurement []  
stop\_pos, stop\_neg:         signal that controls end of measurement []

### Description

This probe measures signals such as minimum, maximum, average, peak-to-peak, root mean square, standard deviation of the output, and start signals within a measuring window. It also gives a correlation coefficient between output and start signals.

### Instance Parameters

start\_arg = argument value that starts measurements  
stop\_arg = argument value that stops measurements  
start\_td, stop\_td = signal delays [s]  
start\_val, stop\_val = signal value that starts/end measurement []  
start\_count, stop\_count = number of signal values that starts/end measurement  
start\_mode = one of starting/stopping modes []  
    arg—argument value (simulation time)  
    rise—crossing of the signal value on rise  
    fall—crossing of the signal value on fall  
    crossing—any crossing of the signal value  
stop\_mode = one of starting/stopping modes []  
    arg—argument value (simulation time)  
    rise—crossing of the signal value on rise

## Cadence Verilog-A Language Reference

### Sample Model Library

---

`fall`—crossing of the signal value on fall

`crossing`—any crossing of the signal value



## Voltage Meter

### Terminals

`vp`, `vn`: terminals [V,A]

`vout`: measured voltage [V,A]

### Description

Measures the voltage between two of its nodes. It has two modes: rms (root-mean-squared) and absolute.

The measurement is passed through a first-order filter with bandwidth `bw` before being written to a file and appearing at `vout`. This is useful when doing rms measurements. If `bw` is set to zero, no filtering is done.

### Instance Parameters

`mtype` = type of voltage measurement; absolute or rms []

`bw` = bw of output filter (a first-order filter) [Hz]

`log_to_file` = whether to log the results to a file; yes or no []

`filename` = the name of the file in which the results are logged []

## Z (Impedance) Meter

### Terminals

`iin`: input for current passing through the meter [V,A]

`vp_iout`: positive voltage-sensing terminal and output for current passing through the meter [V,A]

`vn`: negative voltage sensing terminal [V,A]

`zout`: measured impedance converted to a voltage [Ohms]

### Description

To measure the impedance across a 2-port device, this meter should be placed in the netlist so that the current flowing into the device passes between `iin` and `vp_iout` first, that `vp_iout` is connected to the positive terminal of the device, and that `vn` is connected to the negative terminal of the device.

The impedance is calculated by finding the rms values of the current and voltage first and filtering them with a first-order filter of bandwidth `bw`. The impedance is the ratio of these filtered `Irms` and `Vrms` values. The purpose of the filtering is to remove ripple.

Cadence recommends that `bw` be set to a low value to produce accurate measurements and that at least 10 input AC cycles be allowed before the `zmeter` is considered settled. Also allow time for the filters to settle.

The time step size should also be kept small to increase accuracy.

This meter is nonintrusive—that is, it does not drive current in the device being measured. However to work it requires that something else drives current through the device.

### Instance Parameters

`bw` = bw of rms filters (a first-order filter) [Hz]

`log_to_file` = whether to log the results to a file; `yes` or `no` []

`filename` = the name of the file in which the results are logged []

## Mechanical Systems

### Gearbox

#### Terminals

wshaft1: shaft of the first gear [rad/s, Nm]

wshaft2: shaft of the second gear [rad/s, Nm]

#### Description

This is a model of two intermeshed gears.

#### Instance Parameters

radius1 = radius of first gear [m]

radius2 = radius of second gear [m]

inertia1 = inertia of first gear [Nms/rad]

inertia2 = inertia of second gear [Nms/rad]

## **Mechanical Damper**

### **Terminals**

posp, posn:            terminals [m, N]

### **Instance Parameters**

d = friction coefficient [N/m]

## **Mechanical Mass**

### **Terminal**

`posin:` terminal [m, N]

### **Instance Parameters**

`m` = mass [kg]

`gravity` = whether gravity acting on the direction of movement of mass []

## **Mechanical Restrainer**

### **Terminals**

`posp, posn:`      terminals [m, N]

### **Description**

Limits extension of the nodes to which it is attached.

### **Instance Parameters**

`minl` = minimum extension [m]

`maxl` = maximum extension [m]

## **Road**

### **Terminal**

`posin:`     terminal [m, N]

### **Description**

This is a model of a road with bumps.

### **Instance Parameters**

`height` = height of bumps [m]

`length` = length of bumps [m]

`speed` = speed [m/s]

`distance` = distance to first bump [m]

## **Mechanical Spring**

### **Terminals**

posp, posn:      terminals [m, N]

### **Instance Parameters**

k = spring constant [N/m]

l = length of the spring [m]



## Wheel

### Terminals

`posp, posn:`      terminals [m, N]

### Description

This is a model of a bearing wheel on a fixed surface.

### Instance Parameters

`height` = height of the wheel [m]

## Mixed-Signal Components

### Analog-to-Digital Converter, 8-Bit

#### Terminals

`vin:` [V,A]

`vclk:` [V,A]

`vd0..vd7:` data output terminals [V,A]

#### Description

This ADC comprises 8 comparators. An input voltage is compared to half the reference voltage. If the input exceeds it, bit 7 is set and half the reference voltage is subtracted. If not, bit 7 is assigned zero and no voltage is subtracted from the input. Bit 6 is found by doing an equivalent operation comparing double the adjusted input voltage coming from the first comparator with half the reference voltage. Similarly, all the other bits are found.

Mismatch effects in the comparator reference voltages can be modeled setting `mismatch` to a nonzero value. The maximum `mismatch` on a comparator's reference voltage is  $\pm$  `mismatch` percent of that voltage's nominal value.

#### Instance Parameters

`mismatch_fact` = maximum mismatch as a percentage of the average value []

`vlogic_high` = [V]

`vlogic_low` = [V]

`vtrans_clk` = clk high-to-low transition voltage [V]

`vref` = voltage that voltage is done with respect to [V]

`tdel`, `trise`, `tfall` = {usual} [s]

## **Analog-to-Digital Converter, 8-Bit (Ideal)**

### **Terminals**

vin: [V,A]

vclk: [V,A]

vd0..vd7: data output terminals [V,A]

### **Description**

This model is ideal because no mismatch is modeled.

### **Instance Parameters**

t<sub>del</sub>, t<sub>rise</sub>, t<sub>fall</sub> = {usual} [s]

vlogic\_high = [V]

vlogic\_low = [V]

vtrans\_clk = clk high-to-low transition voltage [V]

vref = voltage that voltage is done with respect to [V]

## Decimator

### Terminals

vin: [V,A]

vout: [V,A]

vclk: [V,A]

### Description

Produces a cumulative average of  $N$  samples of `vin`. `vin` is sampled on the positive `vclk` transition. The cumulative average of the previous set of  $N$  samples is output until a new set of  $N$  samples has been captured.

Transfer Function:  $1/N * (1 - Z^{-N}) / (1 - Z^{-1})$

### Instance Parameters

$N$  = oversampling ratio [V]

`vtrans_clk` = transition voltage of the clock [V]

`tdel`, `trise`, `tfall` = {usual} [s]

## Digital-to-Analog Converter, 8-Bit

### Terminals

`vd0..vd7:` data inputs [V,A]

`vout:` [V,A]

### Description

Mismatch effects can be modeled in this DAC by setting `mismatch` to a nonzero value. The maximum mismatch on a bit is  $\pm$ `mismatch` percent of that bit's nominal value.

### Instance Parameters

`vref` = reference voltage for the conversion [V]

`mismatch_fact` = maximum mismatch as a percentage of the average value []

`vtrans` = logic high-to-low transition voltage [V]

`tdel, trise, tfall` = {usual} [s]

## **Digital-to-Analog Converter, 8-Bit (Ideal)**

### **Terminals**

`vd0..vd7:` data inputs [V,A]

`vout:` [V,A]

### **Instance Parameters**

`vref` = reference voltage that conversion is with respect to [V]

`vtrans` = transition voltage between logic high and low [V]

`tdel, trise, tfall` = {usual} [s]

## **Sigma-Delta Converter (first-order)**

### **Terminals**

vin: [V,A]

vclk: [V,A]

vout: [V,A]

### **Description**

This is a model of a first-order sigma-delta analog-to-digital converter.

### **Instance Parameters**

vth = threshold voltage of two-level quantizer [V]

vout\_high = range of sigma-delta is 0-vout\_high [V]

vtrans\_clk = transition of voltage of clock [V]

tdel, trise, tfall = {usual}

## **Sample-and-Hold Amplifier (Ideal)**

### **Terminals**

vin: [V,A]

vclk: [V,A]

vout: [V,A]

### **Instance Parameters**

vtrans\_clk = transition voltage of the clock [V]



## Single Shot

### Terminals

vin:       input terminal [V,A]

vout:      output terminal [V,A]

### Description

This model outputs a logic high pulse of duration `pulse_width` if a positive transition is detected on the input.

### Instance Parameters

`pulse_width` = pulse width [s]

`vlogic_high` = output voltage for high [V]

`vlogic_low` = output voltage for low [V]

`vtrans` = voltages above this at input are considered high [V]

`tdel`, `trise`, `tfall` = {usual} [s]

## Switched Capacitor Integrator

### Terminals

vout\_p, vout\_n: output terminals [V,A]

vin\_p, vin\_n: input terminals [V,A]

vphi: switching signal [V,A]

### Instance Parameters

cap\_in = input capacitor value

cap\_fb = feedback capacitor value

vphi\_trans = transition voltage of vphi

## Power Electronics Components

### Full Wave Rectifier, Two Phase

#### Terminals

`vin_top:` input [V,A]

`tfire:` delay after positive zero crossing of each phase before phase  
rectifier fires [s,A]

`vout:` rectified output voltage [V,A]

#### Instance Parameters

`ihold` = holding current (minimum current for rectifier to work) [A]

`switch_time` = maximum amount of time to spend attempting switch-on [s]

`vdrop_rect` = total rectification voltage drop [V]

## Half Wave Rectifier, Two Phase

### Terminals

`vin_top:`     input [V,A]

`tfire:`     delay after positive zero crossing of each phase before phase  
          rectifier fires [s,A]

`vout:`     rectified output voltage [V,A]

### Instance Parameters

`ihold` = holding current (minimum current for rectifier to work) [A]

`switch_time` = maximum amount of time to spend attempting switch-on [s]

`vdrop_rect` = total rectification voltage drop [V]

## Thyristor

### Terminals

vanode:     anode [V,A]

vcathode:   cathode [V,A]

vgate:     gate [V,A]

### Instance Parameters

iturn\_on = thyristor gate triggering current [A]

ihold = thyristor hold current [A]

von = thyristor on voltage [V]

## Semiconductor Components

### Diode

#### Terminals

vanode: anode voltage [V,A]

vcathode: cathode voltage [V,A]

#### Description

This model is of a diode based on the Shockley equation.

#### Instance Parameters

$i_s$  = saturation current with negative bias [A]

## **MOS Transistor (Level 1)**

### **Terminals**

vdrain: drain [V,A]

vgate: gate [V,A]

vsource: source [V,A]

vbody: body [V,A]

### **Description**

This model is of a basic, level-1, Schichmann-Hodges style model of a MOSFET transistor.

### **Instance Parameters**

width = [m]

length = [m]

vto = threshold voltage [V]

gamma = bulk threshold []

phi = bulk junction potential [V]

lambda = channel length modulation []

tox = oxide thickness []

u0 = transconductance factor []

xj = metallurgical junction depth []

is = saturation current []

cj = bulk junction capacitance [F]

vj = bulk junction voltage [V]

mj = bulk grading coefficient []

## Cadence Verilog-A Language Reference

### Sample Model Library

---

$f_c$  = forward bias capacitance factor []

$\tau$  = parasitic diode factor []

$c_{gbo}$  = gate-bulk overlap capacitance [F]

$c_{gso}$  = gate-source overlap capacitance [F]

$c_{gdo}$  = gate-drain overlap capacitance [F]

$dev\_type$  = the type of MOSFET used []



## **MOS Thin-Film Transistor**

### **Terminals**

`vdrain:` drain terminal [V,A]  
`vgate_front:` front gate terminal [V,A]  
`vsource:` source terminal [V,A]  
`vgate_back:` back gate terminal [V,A]

### **Description**

This model is of a silicon-on-insulator thin-film transistor.

This is a model of a fully depleted back surface thin-film transistor MOSFET model. No short-channel effects.

### **Instance Parameters**

`length = length []`  
`width = width []`  
`toxf = oxide thickness [m]`  
`toxback = oxide thickness [m]`  
`nsub = [cm-3]`  
`ngate = [cm-3]`  
`nbody = [cm-3]`  
`tb = [m]`  
`u0 = []`  
`lambda = channel length modulation factor []`  
`dev_type = dev_type []`

## **N JFET Transistor**

### **Terminals**

`vdrain:` drain voltage [V,A]

`vgate:` gate voltage [V,A]

`vsource:` source voltage [V,A]

### **Description**

This is a model of an n-channel, junction field-effect transistor.

### **Instance Parameters**

`area` = area []

`vto` = threshold voltage [V]

`beta` = gain []

`lambda` = output conductance factor []

`is` = saturation current []

`gmin` = minimal conductance []

`cjs` = gate-source junction capacitance [F]

`cgd` = gate-drain junction capacitance [F]

`m` = emission coefficient []

`phi` = gate junction barrier potential []

`fc` = forward bias capacitance factor []

## NPN Bipolar Junction Transistor

### Terminals

vcoll: collector [V,A]  
vbase: base [V,A]  
vemit: emitter [V,A]  
vsubs: substrate [V,A]

### Description

This is a gummel-poon style npn bjt model.

### Instance Parameters

area = cross-section area  
is = saturation current []  
ise = base-emitter leakage current []  
isc = base-collector leakage current []  
bf = beta forward []  
br = beta reverse []  
nf = forward emission coefficient []  
nr = reverse emission coefficient []  
ne = b-e leakage emission coefficient []  
nc = b-c leakage emission coefficient []  
vaf = forward Early voltage [V]  
var = reverse Early voltage [V]  
ikf = forward knee current [A]

## Cadence Verilog-A Language Reference

### Sample Model Library

---

$i_{kr}$  = reverse knee current [A]  
 $c_{je}$  = capacitance, base-emitter junction [F]  
 $v_{je}$  = voltage, base-emitter junction [V]  
 $m_{je}$  = b-e grading exponential factor []  
 $c_{jc}$  = capacitance, base-collector junction [F]  
 $v_{jc}$  = voltage, base-collector junction [V]  
 $m_{jc}$  = b-c grading exponential factor []  
 $c_{js}$  = capacitance, collector-substrate junction [F]  
 $v_{js}$  = voltage, collector-substrate junction [V]  
 $m_{js}$  = c-s grading exponential factor []  
 $f_c$  = forward bias capacitance factor []  
 $t_f$  = ideal forward transit time [s]  
 $x_{tf}$  =  $t_f$  bias coefficient []  
 $v_{tf}$  =  $t_f$ - $v_{bc}$  dependence voltage [V]  
 $i_{tf}$  = high current factor []  
 $t_r$  = reverse diffusion capacitance [s]

## Schottky Diode

### Terminals

vanode: anode voltage [V,A]

vcathode: cathode voltage [V,A]

### Description

This model is of a diode based on the Schockley equation.

### Instance Parameters

area = area of junction []

is = saturation current []

n = emission coefficient []

cj0 = zero-bias junction capacitance [F]

m = grading coefficient []

phi = body potential [V]

fc = forward bias capacitance [F]

tt = transit time [s]

bv = reverse breakdown voltage [V]

rs = series resistance [Ohms]

gmin = minimal conductance [Mhos]

## Telecommunications Components

### AM Demodulator

#### Terminals

`vin`: AM RF input signal [V,A]

`vout`: demodulated signal [V,A]

#### Description

Demodulates the signal in `vin` and outputs it as `vout`.

Consists of four stages in series:

1. RF amp amplifier
2. Detector stage (full wave rectifier)
3. AF filters stage is a low-pass filter that extracts the AF signal—has gain of one, and two poles at `af_wn` [rad/s]
4. AF amp stage amplifies by `af_gain` and adds `af_lev_shift`

#### Instance Parameters

`rf_gain` = gain of RF (radio frequency) stage []

`af_wn` = location of both AF (audio frequency) filter poles [rad/s]

`af_gain` = gain of the audio amplifier []

`af_lev_shift` = added to AF signal after amplification and filtering [V]

## AM Modulator

### Terminals

vin: input signal [V,A]

vout: modulated signal [V,A]

### Description

vin is limited to the range between vin\_max and vin\_min. It is also scaled so that it lies within the +/-1 range. This produces vin\_adjusted. vout is given by the following formula:

$$\text{vout} = \text{unmod\_amp} * (1 + \text{mod\_depth} * \text{vin\_adjusted}) * \cos(2 * \text{PI} * \text{f\_carrier} * \text{time})$$

### Instance Parameters

f\_carrier = carrier frequency [Hz]

vin\_max = maximum input signal [V]

vin\_min = minimum input signal [V]

mod\_depth = modulation depth []

unmod\_amp = unmodulation carrier amplitude [V]

## **Attenuator**

### **Terminals**

`vin:` AM input signal [V,A]

`vout:` rectified AM signal [V,A]

### **Description**

`vout` is attenuated by `attenuation`.

### **Instance Parameters**

`attenuation = 20log10 attenuation [dB]`



## Audio Source

### Terminals

vin: [V,A]

vout: [V,A]

### Description

This model synthesizes an audio source. Its output is the sum of 4 sinusoidal sources.

### Instance Parameters

amp1 = amplitude of the first sinusoid [V]

amp2 = amplitude of the second sinusoid [V]

amp3 = amplitude of the third sinusoid [V]

amp4 = amplitude of the fourth sinusoid [V]

freq1 = frequency of the first sinusoid [Hz]

freq2 = frequency of the second sinusoid [Hz]

freq3 = frequency of the third sinusoid [Hz]

freq4 = frequency of the fourth sinusoid [Hz]

## Bit Error Rate Calculator

### Terminals

vin1: [V,A]

vin2: [V,A]

### Description

This model compares the two input signals  $t_{start} + t_{period}/2$  and every  $t_{period}$  seconds later. At the end of the simulation, it prints the bit error rate, which is the number of errors found divided by the number of bits compared.

### Instance Parameters

$t_{start}$  = when to start measuring [s]

$t_{period}$  = how often to compare bits [s]

$v_{trans}$  = voltages above this at input are considered high [V]

## Charge Pump

### Terminals

`vout`: output terminal from which charge pumped/sucked [V,A]

`vsrc`: source terminal from which charge sourced/sunk [V,A]

`siginc, sigdec`: Logic signal that controls charge pump operation [V,A]

### Description

This model can source or sink a fixed current, `iamp`. Its mode depends on the values of `siginc` and `sigdec`;

When `siginc > vtrans`, `iamp` amps are pumped from the output. When `sigdec > vtrans`, `iamp` amps are sucked into the output. When both `siginc` and `sigdec` are in the same state, no current is sucked/pumped.

### Instance Parameters

`iamp` = charging current magnitude [A]

`vtrans` = voltages above this at input are considered high [V]

`tdel, trise, tfall` = {usual} [s]

## **Code Generator, 2-Bit**

### **Terminals**

vout0, vout1:     output bits [V,A]

### **Description**

Generates a pair of random binary signals.

### **Instance Parameters**

seed = random seed

tperiod = period of output code [s]

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

tdel, trise, tfall = {usual} [s]

## Code Generator, 4-Bit

### Terminals

vout\_b0-3:     output bits [V,A]

### Description

This model is of a random 4-bit code generator.

This model outputs a different, randomly generated, 4-bit code every `tperiod` seconds.

### Instance Parameters

`tperiod` = period of the code generation [s]

`vlogic_high` = output voltage for high [V]

`vlogic_low` = output voltage for low [V]

`tdel, trise, tfall` = {usual} [s]

## Decider

### Terminals

vin: [V,A]

vout: [V,A]

### Description

This model samples this input signal a number of times and outputs the most likely value of the binary data contained in the signal.

A decision on what data is contained in the input is made each  $t_{\text{period}}$ . During each decision period, a sample of the input is taken each  $t_{\text{sample}}$ . A count of the number of samples with values greater than  $(v_{\text{logic\_high}} + v_{\text{logic\_low}})/2$  is kept. If at the end of the period, this count is greater than half the number of samples taken, a logic 1 is output. If it is less than half the number of samples,  $v_{\text{logic\_low}}$  is output. Otherwise, the output is  $(v_{\text{logic\_high}} + v_{\text{logic\_low}})/2$ .

The sampling starts at  $t_{\text{start}}$ .

### Instance Parameters

$t_{\text{period}}$  = period of binary data being extracted [s]

$t_{\text{sample}}$  = sampling period [s]

$v_{\text{logic\_high}}$  = output voltage for high [V]

$v_{\text{logic\_low}}$  = output voltage for low [V]

$t_{\text{start}}$  = time at which to start sampling [s]

$t_{\text{del}}, t_{\text{rise}}, t_{\text{fall}}$  = {usual} [s]

## Digital Phase Locked Loop (PLL)

### Terminals

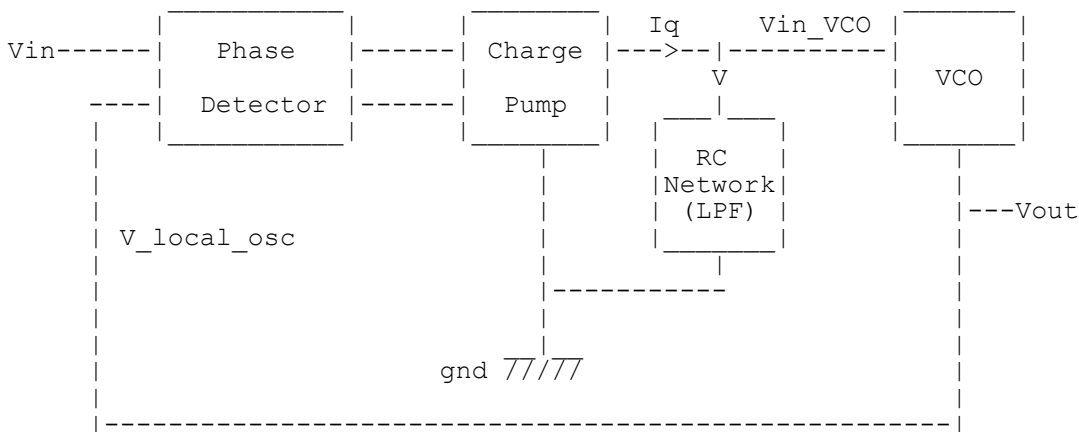
vin: [V,A]

vout: [V,A]

### Description

The model comprises a number of submodels: digital phase detector, a charge pump, a low-pass filter (LPF), and a digital voltage-controlled oscillator (VCO).

They are arranged in the following way:



### Instance Parameters

pump\_iamp = amplitude of the charge pump's output current [A]

vco\_cen\_freq = center frequency of the VCO [Hz]

vco\_gain = the gain of the VCO []

lpf\_zero\_freq = zero frequency of LPF (low-pass filter) [Hz]

lpf\_pole\_freq = pole frequency of LPF [Hz]

lpf\_r\_nom = nominal resistance of RC network implementing LPF

## Digital Voltage-Controlled Oscillator

### Terminals

vin: [V,A]

vout: [V,A]

### Description

The output is a square wave with instantaneous frequency:

$$\text{center\_freq} + \text{vco\_gain} * \text{vin}$$

### Instance Parameters

center\_freq = center frequency of oscillation frequency when  $\text{vin} = 0$  [Hz]

vco\_gain = oscillator conversion gain [Hz/volt]

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

t<sub>del</sub>, t<sub>rise</sub>, t<sub>fall</sub> = {usual} [s]



## FM Demodulator

### Terminals

`vin`: FM RF input signal [V,A]

`vout`: demodulated signal [V,A]

### Description

Demodulates the signal in `vin` and outputs it as `vout`.

Consists of four stages in series:

1. RF amp stage amplifiers `vin`
2. Detector stage is a phase locked loop (PLL)
3. AF filters stage is a low-pass filter that extracts the AF signal. The filter has gain of one, and two poles at `af_wn` [rad/s]
4. AF amp stage amplifies by `af_gain` and adds `af_lev_shift`.

### Instance Parameters

`rf_gain` = gain of RF (radio frequency) stage []

`pll_out_bw` = bandwidth of PLL output filter [Hz]

`pll_vco_gain` = gain of the PLL's VCO []

`pll_vco_cf` = the center frequency of the PLLs [Hz]

`af_wn` = location of both AF (audio frequency) filter poles [Hz]

`af_gain` = gain of the audio amplifier []

`af_lev_shift` = added to AF signal after amplification and filtering [V]

## **FM Modulator**

### **Terminals**

`vin:` input signal [V,A]

`vout:` modulated signal [V,A]

### **Description**

`vout = amp * sin (phase)`

where `phase = integ (2 * PI * f_carrier + vin_gain * vin)`

### **Instance Parameters**

`f_carrier` = carrier frequency [Hz]

`amp` = amplitude of the FM modulator output []

`vin_gain` = amplification of `vin_signal` before it is used to modulate the FM carrier signal []

## Frequency-Phase Detector

### Terminals

`vin_if`: signal whose phase is being detected [V,A]

`vin_lo`: signal from local oscillator [V,A]

`sigout_inc`: logic signal to control charge pump [V,A]

`sigout_dec`: logic signal to control charge pump [V,A]

### Description

The `freq_ph_detector` can have three states: `behind`, `ahead`, and `same`. The specific state is determined by the positive-going transitions of the signals `vin_if` and `vin_lo`.

Positive transitions on `vin_if` causes the state to become the next higher state unless the state is already `ahead`.

Positive transitions on `vin_lo` cause the state to become the next lower state unless the state is already `behind`.

The output depends on the state the detector is in:

`ahead => sigout_inc = high, sigout_dec = low`

`same => sigout_inc = high, sigout_dec = high`

`behind => sigout_inc = low, sigout_dec = high`

The output signals are expected to be used by a `charge_pump`.

### Instance Parameters

`vlogic_high` = output voltage for high [V]

`vlogic_low` = output voltage for low [V]

`vtrans` = voltages above this at input are considered high [V]

`tdel, trise, tfall` = {usual} [s]

## **Mixer**

### **Terminals**

vin1, vin2: [V,A]

vout: [V,A]

### **Description**

$vout = gain * vin1 * vin2$

### **Instance Parameters**

gain = gain of mixer []

## Noise Source

### Terminals

vin: [V,A]

vout: [V,A]

### Description

This is an approximate white noise source.

**Note:** It is *not* a true white source because its output changes every time step and the time step is dependent on the behavior of the circuit.

### Instance Parameters

amp = amplitude of the output signal about 0 [V]

## PCM Demodulator, 8-Bit

### Terminals

`vin`: input signal [V,A]

`vout`: demodulated signal [V,A]

### Description

The PCM demodulator samples `vin` at `bit_rate` [Hz] starting at `tstart + 0.5/bit_rate`. Each set of 8 samples is considered a binary word, and these sets are converted to an output voltage using a linear 8-bit binary code with 0 representing `vin_min` and 255 representing `vin_max`. The first bit received is the LSB, bit 0; the last bit received is the MSB, bit 7.

The output rate is `bit_rate/8`.

### Instance Parameters

`freq_sample` = sample frequency [Hz]

`tstart` = when to start sampling [s]

`vout_min` = minimum input voltage [V]

`vout_max` = maximum input voltage [V]

`vtrans` = voltages above this at input are considered high [V]

`tdel, trise, tfall` = {usual} [s]

## PCM Modulator, 8-Bit

### Terminals

`vin`: input signal [V,A]

`vout`: modulated signal [V,A]

### Description

The PCM modulator samples `vin` at a `sample_freq` [Hz] starting at `tstart`. Once a sample has been obtained, it is converted to a linear 8-bit binary code with 0 representing `vin_min` and 255 representing `vin_max`.

The bits are in the code and are sequentially put through `vout` at a rate 8 times `sample_freq` with `vlogic_high` signifying a 1 and `vlogic_low` signifying a 0. The first bit transmitted is the LSB, bit 0; the last bit transmitted is the MSB, bit 7.

Clipping occurs when the input is outside `vin_min` and `vin_max`.

### Instance Parameters

`sample_freq` = sample frequency [Hz]

`tstart` = when to start sampling [s]

`vin_min` = minimum input voltage [V]

`vin_max` = maximum input voltage [V]

`vlogic_high` = output voltage for high [V]

`vlogic_low` = output voltage for low [V]

`tdel, trise, tfall` = {usual} [s]

## Phase Detector

### Terminals

`vlocal_osc`: local oscillator voltage [V,A]

`vin_rf`: PLL radio frequency input voltage [V,A]

`vif`: intermediate frequency output voltage [V,A]

### Instance Parameters

`gain` = gain of detector []

`mtype` = type of phase detection to be used; chopper or multiplier []



## Phase Locked Loop

### Terminals

vlocal\_osc: local oscillator voltage [V,A]

vin\_rf: PLL radio frequency input voltage [V,A]

vout: voltage proportional to the frequency being locked onto [V,A]

vout\_ph\_det: output of the phase detector [V,A]

### Instance Parameters

vco\_gain = gain of VCO cell [Hz/V]

vco\_center\_freq = VCO oscillation frequency [Hz]

phase\_detect\_type = type of phase detection cell to be used []

vout\_filt\_bandwidth = bandwidth of the low-pass filter on output [Hz]

## PM Demodulator

### Terminals

`vin`: PM RF input signal [V,A]

`vout`: demodulated signal [V,A]

### Description

Demodulates the signal in `vin` and outputs it as `vout`.

Consists of four stages in series:

1. RF amp stage amplifiers `vin`.
2. Detector stage is a phase locked loop (PLL)—the phase detector output is tapped.
3. AF filters stage is a low-pass filter that extracts the AF signal—has gain of one, and two poles at `af_wn` [rad/s].
4. AF amp stage amplifies by `af_gain` and adds `af_lev_shift`.

### Instance Parameters

`rf_gain` = gain of RF (radio frequency) stage []

`pll_out_bw` = bandwidth of PLL output filter [Hz]

`pll_vco_gain` = gain of the PLL's VCO []

`pll_vco_cf` = the center frequency of the PLLs [Hz]

`af_wn` = location of both AF (audio frequency) filter poles [Hz]

`af_gain` = gain of the audio amplifier []

`af_lev_shift` = added to AF signal after amplification and filtering [V]

## PM Modulator

### Terminals

vin: input signal [V,A]

vout: modulated signal [V,A]

### Description

$vout = amp * \sin(2 * PI * f\_carrier * time + phase\_max * vin\_adjusted)$

where `vin_adjusted` is scaled version of `vin` that lies within the +/-1 range.

Before scaling, `vin` is limited to the range between `vin_max` and `vin_min` by clipping.

### Instance Parameters

`f_carrier` = carrier frequency [Hz]

`amp` = amplitude of the PM modulator output []

`vin_max` = maximum acceptable input (clipping occurs above this) [V]

`vin_min` = minimum acceptable input (clipping occurs above this) [V]

`phase_max` = the phase shift produced when the modulating signal is at `vin_max` [rad]

## QAM 16-ary Demodulator

### Terminals

vin:           input [V,A]

vout\_bit[0-4]:    demodulated codes [V,A]

### Description

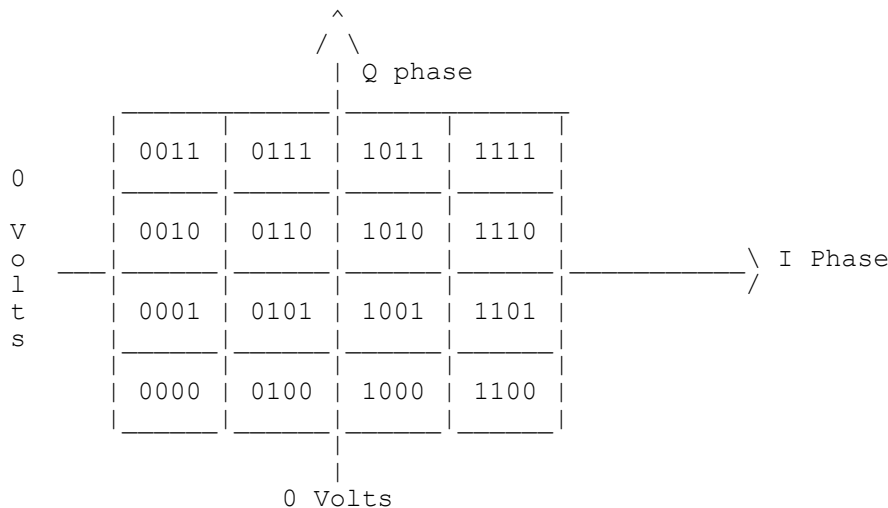
This model is of a QPSK (quadrature phase shift key) modulator.

Demodulates a 16ary encoded QAM signal by separately sampling the input signal at 90 degrees (q-phase) and 180 degrees (i-phase).

This model does not contain a dynamic synchronizing mechanism for ensuring that sampling occurs at the correct time points. Synchronizing can be statically adjusted by changing `tstart`. `tstart` should correspond to when the input QAM signal is at 0 degrees.

The i-phase contains the two MSBs. The q-phase contains the two LSBs.

The constellation diagram representing this relationship follows.



Each code box is `vbox_width` volts wide.

### Instance Parameters

`freq` = demodulation frequency [Hz]

## Cadence Verilog-A Language Reference

### Sample Model Library

---

`vbox_width` = width of modulation code box in constellation diagram [V]

`vlogic_high` = output voltage for high [V]

`vlogic_low` = output voltage for low [V]

`tdel`, `trise`, `tfall` = {usual} [s]

## Quadrature Amplitude 16-ary Modulator

### Terminals

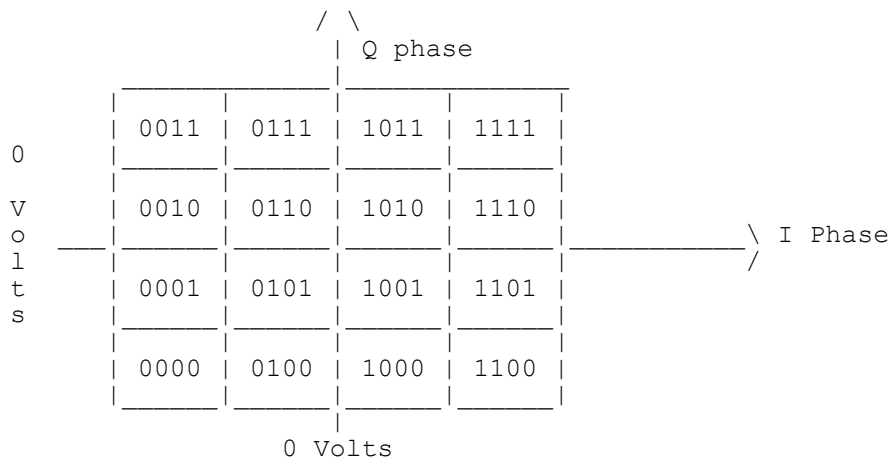
vin\_b[0-3]: bits of input code [V,A]

vout: modulated output [V,A]

### Description

This model does 16 value (4-Bit) QAM.

It encodes the MSBs on the i-phase and the LSBs on the q-phase. Its constellation diagram can be represented as



The two MSBs are encoded on the i-phase. The two LSBs are encoded on the q-phase.

The modulating formula is  $V_{out} = i\_phase * \cos(\omega t) + q\_phase * \sin(\omega t)$

$i\_phase$  and  $q\_phase$  vary between  $-phase\_ampl$  and  $phase\_ampl$ .

### Instance Parameters

freq = modulation frequency [Hz]

phase\_ampl = amplitude of the i-phase and q-phase signals [V]

vtrans = voltages above this at input are considered high [V]

tdel, trise, tfall = {usual} [s]

## QPSK Demodulator

### Terminals

`vin:`       input [V,A]  
`vout_i:`     i-phase output [V,A]  
`vout_q:`     q-phase output [V,A]

### Description

Does a QPSK demodulation on the input signal. It does not contain a dynamic synchronizing mechanism. Synchronizing can be adjusted by changing `tstart`.

Detection works by separately sampling the i-phase of `vin` and the q-phase of `vin` at `freq` Hz and 90 degrees out of phase. The first i-phase sample is done at `tstart + 0.5/freq`, the next  $1/freq$  seconds later, etc. Similarly, the first q-phase sample is done at `tstart + 0.25/freq`, the next  $1/freq$  seconds later, and so on.

For the i-phase, a high is detected if the sample  $< -vthresh$ . For the q-phase, a high is detected if the sample  $> vthresh$ .

### Instance Parameters

`freq` = demodulation frequency [Hz]  
`vthresh` = threshold detection voltage [V]  
`vlogic_high` = output voltage for high [V]  
`vlogic_low` = output voltage for low [V]  
`tstart` = time at which demodulation starts [s]  
`tdel, trise, tfall` = {usual} [s]

## **QPSK Modulator**

### **Terminals**

`vin_i, vin_q:` quadrature inputs [V,A]

`vout:` modulator output [V,A]

### **Description**

This takes two sampled quadrature inputs and does QPSK modulation on them.

### **Instance Parameters**

`freq` = modulation frequency [Hz]

`amp` = modulator amplitude [V]

`vtrans` = voltages above this at input are considered high [V]

`tdel, trise, tfall` = {usual} [s]



## Random Bit Stream Generator

### Terminal

vout: [V,A]

### Description

This model generates a random stream of bits.

### Instance Parameters

tperiod = period of stream [s]

seed = random number seed []

vlogic\_high = output voltage for high [V]

vlogic\_low = output voltage for low [V]

tdel, trise, tfall = {usual} [s]

## Transmission Channel

### Terminals

`vin`: AM input signal [V,A]

`vout`: rectified AM signal [V,A]

### Description

`vin` has `noise_amp` noise added to it and the resultant is attenuated by `attenuation` [dB].

### Instance Parameters

`attenuation` =  $20\log_{10}$  attenuation [dB]

`noise_amp` = amplitude of noise added to `vin` *before* attenuation [V]

## **Voltage-Controlled Oscillator**

### **Terminals**

`vin`: oscillation-controlling voltage [V,A]

`vout`: [V,A]

### **Instance Parameters**

`amp` = amplitude of the output signal [V]

`center_freq` = center frequency of oscillation frequency when `vin = 0` [Hz]

`vco_gain` = oscillator conversion gain [Hz/volt]

**Cadence Verilog-A Language Reference**  
Sample Model Library

---

---

## Understanding Error Messages

---

When you use the Cadence® Verilog®-A language within the Cadence analog design environment, the compiler and simulator send error messages to the verilog Parser Error/Warnings window or to the Command Interpretation Window (CIW) and the log file. When you run Verilog-A outside the Cadence analog design environment, error messages are sent to the standard output.

The following module contains an error in the line containing the first `$strobe` statement. The variable `xx` is referenced there but has not been declared.

```
`include "disciplines.vams"

module prove_v(vin, vgn) ;
input vin, vgn ;
electrical vin, vgn ;

analog begin
    $strobe("%f, %f", xx, V(vin,vgn));    // ERROR! xx not declared
    $strobe("lo");
end
endmodule
```

Verilog-A produces the following error message when it attempts to compile module `prove_v`.

```
Error found by spectre during AHDL read-in.
"unknown_id.va", line 8: "$strobe("%f, %f", xx,<--?
    V(vin,vgn));"
"unknown_id.va", line 8: Error: undeclared symbol: xx.
"unknown_id.va", line 8: Error: argument #3 does not
    match %f in argument #1; real expected.
```

There are two main forms of error messages: the token indication form and the description form. In the example above, the first error message is a token indication message. The token indicator `<--?` points to the first token on a line where Verilog-A finds an error.

The other error messages are description error messages. The first description error message corresponds to the token indication error message.

For some errors, Verilog-A gives the message `syntax error`. This means that the compiler is unable to determine the exact cause of the error. To find the problem, look where the token

## Cadence Verilog-A Language Reference

### Understanding Error Messages

---

indicator is pointing. Look also at the preceding line to see if there is anything wrong with it, such as a missing semicolon. For example, the following module is missing a semicolon in line 9.

```
`include "disciplines.vams"

module probe_v2(vout, vin_p, vin_n) ;
input vin_p, vin_n ;
output vout ;
electrical vout, vin_p, vin_n ;

analog begin
    $strobe("hi") // ERROR! Missing semicolon.
    $strobe("lo") ;
    V(vout) <+ V(vin_p,vin_n) ;
end

endmodule
```

However, the problem is reported as a syntax error in line 10.

```
Error found by spectre during AHDL read-in.
"miss_semi1.va", line 10: "$<<--? strobe("lo");"
"miss_semi1.va", line 10: Error: syntax error
```

If the compiler reports another error before a syntax error, fix the first error and try to compile the Verilog-A file again. Subsequent syntax errors might actually be a result of an initial error. A single mistake can result in a number of error messages.

Token indication error messages report only one error per line. The compiler, however, can generate multiple description error messages about other errors on that line.

---

## Getting Ready to Simulate

---

The following topics apply to setting up a simulation of a design you create using the Cadence® Verilog®-A language.

- [Creating a Verilog-A Module Description](#) on page 520
- [Creating a Spectre Netlist File](#) on page 523
- [Modifying Absolute Tolerances](#) on page 527
- [Using the Compiled C Code Flow](#) on page 531
- [Using Verilog-A Compact Models to Increase Simulation Speed](#) on page 536
  - [Noticing Differences When You Use the compact\\_module Attribute](#) on page 537
  - [Specifying Instance and Model Parameters for a Verilog-A Compact Model](#) on page 537
  - [Model Binning for Verilog-A Compact Models](#) on page 538
- [Ignoring the State of a Verilog-A Module for RF Simulation](#) on page 539
- [Ignoring the State of a Verilog-A Local Variable for RF Simulation](#) on page 540

Except as noted, these topics assume you are working outside the Cadence analog design environment. For information on working inside the design environment, see [Chapter 13, “Using Verilog-A in the Cadence Analog Design Environment.”](#)

## Creating a Verilog-A Module Description

Use a text editor to create the following file, which contains a Verilog-A description of a simple resistor. Save the file with the name `res.va`. Alternatively, you can copy the example from the sample model library

```
your_install_dir/tools/dfII/samples/artist/spectreHDL/Verilog-A/basic/res.va
```

Lines beginning with `//` are comment lines and are ignored by the simulator.

```
// res.va, a simple resistor

`include "disciplines.vams"
`include "constants.vams"

module res(vp, vn);
  inout vp, vn;
  electrical vp, vn;
  parameter real r = 0;

  analog
    V(vp, vn) <+ r*I(vp, vn);
endmodule
```

### See also

- [File Extension .va](#) on page 520
- [`include Compiler Directive](#) on page 520
  - [Absolute Paths](#) on page 521
  - [Relative Paths](#) on page 521
  - [Simple File Name](#) on page 522
- [CDS\\_MMSIM\\_VERILOGA Macro](#) on page 523

### File Extension .va

The simulator expects all files containing Verilog-A modules to have the file extension `.va`. The simulator uses the file extension to identify which language is used in a file.

### `include Compiler Directive

With the Verilog-A ``include` compiler directive, you can include another file in the current file. The compiler copies the included file into the current file and applies any compiler directives currently in effect to the included file. If the included file itself contains any compiler



## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

directives, the compiler applies them to the rest of the file that is doing the including. For additional information, see [“Including Files at Compilation Time”](#) on page 242.

With the file name on the ``include` directive, you can specify a full or relative path. As explained in the topics that follow, the path and file name you specify control where the compiler searches for an included file. See:

- [Absolute Paths](#) on page 521
- [Relative Paths](#) on page 521
- [Simple File Name](#) on page 522

File `res.va`, in the previous example, includes two files: `disciplines.vams` and `constants.vams`. These files are part of the Cadence distribution (installed in `your_install_dir/tools/spectre/etc/ahdl`). The `disciplines.vams` file contains definitions for the standard natures and disciplines. In particular, `disciplines.vams` includes a definition of the `electrical` discipline referenced in `res.va`. If your module, like most Verilog-A modules, uses the standard disciplines, you must include the `disciplines.vams` file.

The `constants.vams` file contains definitions of commonly used mathematical and physical constants such as Pi and Boltzmann’s constant. If your module uses the standard constants, you must include the `constants.vams` file. The module `res` does not use any of the standard constants, so the example includes the `constants.vams` file only for consistency.

### Absolute Paths

If you specify an absolute path (one that starts with `/`), the compiler searches for the include file only in the specified directory. If the file is not in this directory, the compiler issues an error message.

This is an example using an absolute path:

```
`include "/usr/local/include/disciplines.vams"
```

### Relative Paths

A relative path is one that starts with `./`, `../`, or `dir/`, where `dir` is a subdirectory. If you specify a relative path for the ``include` compiler directive, the compiler searches relative to the directory containing the Verilog-A file (`.va` file) that contains the ``include` directive. If the file to be included is not in the directory specified by the relative path, the compiler issues an error message.

If you specify a relative path such as

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

```
`include "./disciplines.vams"
```

the compiler looks only in the directory that contains the file with the ``include` directive.

If you specify a relative path such as

```
`include "../disciplines.vams"
```

the compiler looks only in the parent directory of the `.va` file with the ``include` directive.

The next example illustrates how you might include a capacitor model from a subdirectory that is two levels below the current directory.

```
`include "models/vloga/cap.va"
```

The final relative path example illustrates how you might include a flip-flop module definition located in a sibling directory.

```
`include "../logic/flip_flop.va"
```

### Simple File Name

If you do not specify a path in the file name, the compiler searches three places, in the following order:

1. The directory that contains the file with the ``include` directive
2. The directory specified by the `CDS_VLOGA_INCLUDE` environment variable, if you have set this variable

#### **Important**

AMS Designer does not make use of this setting.

3. The directory specified by

```
your_install_dir/tools/dfII/samples/artist/spectreHDL/include
```

where *your\_install\_dir* is the path to the Cadence installation directory

Usually, this applies when you include the `disciplines.vams` and `constants.vams` files (installed in *your\_install\_dir*/tools/spectre/etc/ahdl). As a result, you generally do not have to worry about the location of these files.

If the file is not in any of these three places, the compiler issues an error message. If the file exists in more than one of these places, the software includes the first one it encounters.

## **CDS\_MMSIM\_VERILOGA Macro**

You can use the predefined CDS\_MMSIM\_VERILOGA macro as follows to create a Verilog-A module that you want to target for different Verilog-A simulators:

```
analog begin
    [statement1]
    [statement2]
'ifdef CDS_MMSIM_VERILOGA
    [statement3] // only MMSIM simulators will pick up this statement
                // (Spectre, UltraSim, AMS Designer)
'else
    [statement4] // MMSIM simulators will not pick up this statement
'endif
end
```

You can also undefine this macro as follows:

```
'undef CDS_MMSIM_VERILOGA
```

## **Creating a Spectre Netlist File**

To use the module defined in `res.va` you must *instantiate* it. To instantiate a module, you prepare a Spectre netlist file that directly or hierarchically creates one or more named instances of the module, instances of other required modules, and any required simulation stimuli and analysis descriptions. In this release of Verilog-A, you must instantiate at least one module directly in the netlist file. Instantiated modules can hierarchically instantiate other modules within themselves by using the support provided by the Verilog-A language. See [Chapter 10, “Instantiating Modules and Primitives,”](#) for more information.

Use a text editor to create the following netlist file. Save the file with the name `res.ckt`. Alternatively, you can copy the example from the sample model library:

```
your_install_dir/tools/dfII/samples/artist/spectreHDL/Verilog-A/basic/test/
res.ckt
```

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

where `your_install_dir` is the path to the Cadence installation directory.

```
// netlist file
// res test circuit
//

global gnd
simulator lang=spectre

ahdl_include "res.va"

i1 in gnd  isource dc=1m
r1 in gnd  res  r=1k

saveNodes options save=allpub

paramSwp dc start=1 stop=1001 param=r dev=r1
```

**Note:** If you copy `res.ckt` from the sample model library, be sure to edit the file and remove the `../` part from the relative path in the `ahdl_include` statement.

The netlist file `res.ckt` includes the Verilog-A description file `res.va` by using the `ahdl_include` statement. When the simulator encounters an `ahdl_include` statement in the netlist file, it looks at the filename extension to determine how to compile the source description. Because of the `.va` file extension, the simulator expects the included file to contain a Verilog-A description and compiles it accordingly.

The `res.ckt` netlist file creates an instance `i1` of a current source and an instance `r1` of a resistor. The current source is an example of a built-in Spectre primitive component. The resistor is an instance of the Verilog-A module that you specified in `res.va`.

The last line in the netlist file tells Spectre to simulate the component behavior as the parameter `r` of instance `r1` sweeps from 1 ohm to 1,001 ohms.

## Including Files in a Netlist

Use the `ahdl_include` Spectre statement to include Verilog-A module description files in a netlist file. The `ahdl_include` statement has the form

```
ahdl_include "filename" [ -master mapped_name ]
```

If `filename` is not in the same directory as the netlist, `filename` must either include the complete path to the module file or be on the path specified in the `-I` option when you start Spectre.

The optional `-master` option allows you to use multiple views of a single module in a circuit. With this option, you can use modules that share the same name whose definitions are in

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

different files. For example, the following two modules have the same name but different definitions in different files.

The file `va_res.va` contains a module definition for `res_va`:

```
`include "discipline.vams"
`include "constants.vams"
module res_va(plus, minus);
  inout plus, minus;
  electrical plus, minus;
  parameter real lr=1;
  parameter real wr=1;
  parameter real rsh=1;
  parameter real dw=1;

  real r;

  analog begin
    r=rsh*lr/(4.0*(wr-2*dw));
    V(plus, minus) <+ r*I(plus, minus);
  end
endmodule
```

The file `another_res.va` contains another module definition for `res_va`:

```
`include "discipline.vams"
`include "constants.vams"
module res_va(plus, minus);
  inout plus, minus;
  electrical plus, minus;
  parameter real lr=1;
  parameter real wr=1;
  parameter real rsh=1;
  parameter real dw=1;

  real r;

  analog begin
    r=rsh*lr/(8*(wr-4*dw));
    V(plus,minus) <+ r*I(plus,minus);
  end
endmodule
```

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

To use both of these modules in a netlist, you map one of them to a different name using the `-master` option, as illustrated by the following netlist file. You can instantiate both the original `res_va` and the mapped `res_va` (`res_va_mapped`).

```
// netlist file
// res test circuit
//
ahdl_include "va_res.va"
ahdl_include "another_res.va" -master res_va_mapped
ar1 1 0 res_va
r2 1 0 res_va_mapped
saveNodes options save=allpub
tranRsp tran start=0 stop=10m
```

**Note:** When you use Verilog-A in the Virtuoso® Analog Design Environment, the software inserts the `-master` switch for you automatically as needed. For example, if you select modules from two different libraries, or have multiple Verilog-A views for a single cell that have the same module name, the netlister automatically maps module names as necessary and adds the `-master` switch to the instantiation.

### Naming Requirements for SPICE-Mode Netlisting

If you want to mix SPICE-mode netlisting (primitive types identified by the first character of the instance name) into the same module definition text file, you must use only lowercase characters in the names of modules, nodes, and parameters.

## Modifying Absolute Tolerances

Verilog-A nature definitions allow you to specify the absolute tolerance (`abstol`) values used by the simulator to determine when convergence occurs during a simulation. The `disciplines.vams` file contains statements that specify default values of `abstol` for the standard natures. You can override these default values, if you wish, by using one of the following two techniques:

- When using Spectre standalone, you can use the ``define` compiler directive in conjunction with the `disciplines.vams` include file.
- When using Spectre in the Cadence analog design environment, you can use Spectre quantities in the netlist file.

### Modifying `abstol` in Standalone Mode

The following text describes how to modify `abstol` for the nature `Voltage` in one place and to have the Verilog-A modules in all your source files use the new `abstol`. This involves specifying the tolerance using a Verilog-A ``define` compiler directive, followed by including the `disciplines.vams` header file, which is then followed by the files containing the module descriptions.

Consider a resistor module specified in the file `my_res.va` and a capacitor module specified in the file `my_cap.va`.

```
// file "my_res.va", a simple resistor

module res(vp, vn);
inout vp, vn;
electrical vp, vn;
parameter real r = 0;
    analog
        V(vp, vn) <+ r*I(vp, vn);
endmodule

// file "my_cap.va", a simple capacitor

module cap(vp, vn);
inout vp, vn;
electrical vp, vn;
parameter real c = 1n;

    analog
        I(vp, vn) <+ c*ddt(V(vp, vn));

endmodule
```

The main instantiating circuit is described in file `my_rc.va`.

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

```
// file "my_rc.va" an rc filter
// this module uses hierarchical instantiation only

`define VOLTAGE_ABSTOL 1e-7
`include "disciplines.vams" // this will use `VOLTAGE_ABSTOL of 1e-7

`include "my_res.va      // include the resistor description
`include "my_cap.va      // include the capacitor description

module my_rc( in, out, gnd );
inout in,out,gnd;
electrical in,out,gnd;
parameter real r=1;
parameter real c=1n;

res #(.r(r)) r1 ( in, out );
cap #(.c(c)) c1 ( out, gnd );

endmodule
```

The ``define` compiler directive in `my_rc.va` sets the `abstol` value that is to be used by the nature `Voltage` (and the electrical discipline) in one place, before the `disciplines.vams` file is included. As a result, the nature `Voltage` is defined with the specified absolute tolerance of `1e-7` when the `disciplines.vams` file is processed. You can override the default absolute tolerances for other natures in the same way.

The descriptions for the resistor and capacitor modules are not given in the file `my_rc.va`, but instead they are included into this file by the ``include` compiler directive. Because the `disciplines.vams` file is included only once, the natures and disciplines it defines are used by both the resistor module and the capacitor module. In this example, both modules use an absolute tolerance for `Voltage` of `1e-7`.

Because modules `res` and `cap` are hierarchically instantiated in module `my_rc.va`, the netlist file `my_rc.ckt` contains only one `ahdl_include` statement.

```
// netlist file
// file "my_rc.ckt", rc_filter test circuit
//

global gnd
simulator lang=spectre

ahdl_include "my_rc.va"

// input voltage to filter
il in gnd vsource type=sine freq=1k

// instantiate an rc filter
f1 in out gnd my_rc r=1k c=1u

// run transient analysis
tranRsp tran start=0 stop=10m
```



## Modifying `abstol` in the Cadence Analog Design Environment

Another way to modify absolute tolerances is to use the Spectre netlist `quantity` statement. A Spectre netlist quantity can be used to specify or modify information about particular types of signals, such as their units, absolute tolerances, and maximum allowed change per Newton iteration. The values specified on a `quantity` statement override any values specified in the `disciplines.vams` include file. For more information, see [“Defining Quantities”](#) on page 264.

Every nature has a corresponding quantity that can be accessed in the Spectre netlist. The name of the quantity is the access function of the nature.

The netlist file `another_rc.ckt` below contains two `ahdl_include` statements. The netlist file also contains a quantity definition that specifies an `abstol` of `1e-7` for the quantity `V`, which corresponds to the `Voltage` nature.

**Note:** When you are working in the Cadence analog design environment, each module file must include the `disciplines.vams` file. If you define a nature or discipline more than once and those definitions have different attributes, the simulator reports an error.

In the following example, the simulator processes the `another_res.va` and `another_cap.va` files separately because they are in separate `ahdl_include` statements. Consequently, each file must contain explicit definitions for the `electrical` discipline. To meet this requirement, both the `another_res.va` source file and the `another_cap.va` source file include the `disciplines.vams` file.

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

Here is the netlist that instantiates the two modules.

```
// netlist file
// file "another_rc.ckt", rc_filter test circuit
//

global gnd
simulator lang=spectre

ahdl_include "another_res.va"
ahdl_include "another_cap.va"

// input voltage to filter
il in gnd vsource type=sine freq=1k

// create the filter using resistor and capacitor
r1 in out another_res r=1k
c1 out gnd another_cap c=1u

// modify the abstol for the Voltage quantity
modifyV quantity name="V" abstol=1e-7

// run transient analysis
tranRsp tran start=0 stop=10m
```

#### File another\_res.va contains

```
// file "another_res.va", a simple resistor
`include "disciplines.vams"

module another_res(vp, vn);
inout vp, vn;
electrical vp, vn;
parameter real r = 0;

    analog
        V(vp, vn) <+ r*I(vp, vn);

endmodule
```

#### File another\_cap.va contains

```
// file "another_cap.va", a simple capacitor
`include "disciplines.vams"

module another_cap(vp, vn);
inout vp, vn;
electrical vp, vn;
parameter real c = 1n;

    analog
        I(vp, vn) <+ c*ddt(V(vp, vn));

endmodule
```

## Using the Compiled C Code Flow

Using the compiled C code flow, the software compiles analog blocks of Verilog-A compact models into shared objects for faster simulation. The shared object contains all the functionality required to simulate the Verilog-A compact model. As long as the Verilog-A file does not change, the compiler does not need to recompile each time you simulate. All netlists that use the shared objects simulate faster. Not only the person who compiles the modules, but other designers can use the shared objects to benefit from improved performance.

### Compiling Verilog-A Compact Models for Reuse

You might have a group in your organization that compiles these models and maintains them in read-only files in a central location for everyone's use.

To compile Verilog-A compact models for reuse, do the following:

1. Set the CDS\_AHDL\_CMI\_SHIPDB\_COPY environment variable to YES.

```
setenv CDS_AHDL_CMI_SHIPDB_COPY YES
```

2. Set the CDS\_AHDL\_CMI\_SHIPDB\_DIR to a directory for the ahdlShipDB. For example:

```
setenv CDS_AHDL_CMI_SHIPDB_DIR /export/shared/objects
```

**Note:** The person who compiles the models must have write access to this directory.

3. Use ahdl\_include statements to include the Verilog-A compact model files.

#### *Important*

There must be at least one instance in the design file of every Verilog-A compact model for which you want to generate a shared object file. The compiler does not generate a shared object file for any module/model that does not have at least one instance in the design file.

4. Run the Spectre circuit simulator.

The program compiles the models and writes shared object files to the specified directory. Designers who include these Verilog-A compact model files can reuse the shared object files without needing the AHDL compiler or GCC.

#### *Important*

You need to create new shared object files whenever a Verilog-A compact model file changes, or whenever you install a new version of the Spectre circuit simulator.

See also

- [Creating and Specifying Compiled C Code Databases](#) on page 533
- [Reusing and Sharing Compiled C Objects](#) on page 534.

## Reusing Verilog-A Shared Objects

To reuse the [Verilog-A shared objects](#), do the following:

1. Set the `CDS_AHDL_CMI_SHIPDB_DIR` to a directory for the `ahdlShipDB`. For example:  

```
setenv CDS_AHDL_CMI_SHIPDB_DIR /export/shared/objects
```
2. Use `ahdl_include` statements to include the Verilog-A compact model files that have corresponding shared object files.
3. Run the Spectre circuit simulator.

The program uses the shared object files in the `ahdlShipDB`.

See also

- [Creating and Specifying Compiled C Code Databases](#) on page 533
- [Reusing and Sharing Compiled C Objects](#) on page 534.

## Controlling the Optimization Level of Compiled C Code Flow

When running the Compiled-C flow, you can use `env CDS_CMI_COMPLEVEL` to control `gcc` optimization level. The `env` value is from 0 to 3 which implies that `gcc` will use level `00 ~ 03` to compile generated `c` files.

The default compilation level (`-O3`) gives better optimization, but requires more memory. It is recommended that `CDS_CMI_COMPLEVEL` is set to lower the compilation level only when the compilation error *gcc out of memory when using -O3 to compile large c file* is received.

## Turning the Compiled C Code Flow Off and On

By default, the software uses compiled C code: The `CDS_AHDL_CMI_ENABLE` environment variable is unset (with a default value of `YES`) and the `-ahdlcom` option is unset (with a default value of 1).

To turn off the compiled C code flow, do one of the following:

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

- Set the `CDS_AHDL_CMI_ENABLE` environment variable to `NO`.

To resume using the compiled C code flow, either unset the `CDS_AHDL_CMI_ENABLE` environment variable or set it to `YES`.

- Set the `spectre -ahdlcom` option to 0 (zero).

Specifying `-ahdlcom 0` results in faster compilation (by turning off the compiled C code flow) but slower simulation. Specifying `-ahdlcom 1` results in slower compilation (by turning on the compiled C code flow) but faster simulation.

**Note:** You can use `-ac` as shorthand for the `-ahdlcom` option.

See also

- [Compiling Verilog-A Compact Models for Reuse](#) on page 531
- [Reusing Verilog-A Shared Objects](#) on page 532
- [Creating and Specifying Compiled C Code Databases](#) on page 533
- [Reusing and Sharing Compiled C Objects](#) on page 534
- [Turning Off Parallel Compilation](#) on page 534

## Creating and Specifying Compiled C Code Databases

The compiled C code flow stores shared objects in a database on disk for the simulator to use: The AHDL simulation database (`ahdlSimDB`). The software creates this database in the current working directory. The name of the database is the root of the design name with a `.ahdlSimDB` extension. For example, if the design name is `top4.scs`, the software creates a database named `top4.ahdlSimDB`.

To specify an alternative location for the `ahdlSimDB`, set the `CDS_AHDL_CMI_SIMDB_DIR` environment variable to the path of a directory. The path must be writable.

To store compiled objects, you use AHDL ship databases (`ahdlShipDBs`). To create such databases, you set the `CDS_AHDL_CMI_SHIPDB_COPY` environment variable to `YES`. When you use this setting, the software creates an `ahdlShipDB` for each Verilog-A file in the directory that contains the Verilog-A file, if the directory is writable. (If the directory is not writable, the software cannot create any `ahdlShipDBs` for the modules in the Verilog-A file.)

If you additionally set the `CDS_AHDL_CMI_SHIPDB_DIR` environment variable to a writable path, the software creates an `ahdlShipDB` there and all the Verilog-A files share it. If the `CDS_AHDL_CMI_SHIPDB_DIR` variable does not specify a writable path or the path does not exist, the software does not create any `ahdlShipDBs` and issues a warning instead.

While looking for already compiled shared objects, the Spectre circuit simulator automatically looks for ahdIShipDBs in the same location as the Verilog-A files. If you set the `CDS_AHDL_CMI_SHIPDB_DIR` environment variable to a particular path, Spectre looks in this path for already-compiled shared objects.

## Reusing and Sharing Compiled C Objects

When you rerun a netlist in the same directory you used before, the software reuses shared objects in the ahdIShipDB automatically.

You can minimize the compilation of shared objects when you run different netlists that share the same Verilog-A files by doing one of the following:

- ▶ If your Verilog-A files are in writable directories, set the `CDS_AHDL_CMI_SHIPDB_COPY` environment variable to `YES`.

The software puts shared objects from the first simulation in the ahdIShipDB that it creates for each Verilog-A file in the same directory as the Verilog-A file it is processing. Subsequent simulations reuse these shared objects.

- ▶ If your Verilog-A files are in read-only directories, set the `CDS_AHDL_CMI_SHIPDB_COPY` environment variable to `YES` and set `CDS_AHDL_CMI_SHIPDB_DIR` to a writable directory.

This directory becomes the sole ahdIShipDB. The software puts shared objects from the first simulation in this ahdIShipDB. Subsequent simulations reuse these shared objects.

To share precompiled objects among different users,

- ▶ Run the simulation once with the `CDS_AHDL_CMI_SHIPDB_COPY` variable set to `YES`.

The software creates an ahdIShipDB for each Verilog-A file in the same directory as the Verilog-A file (provided that the directories containing the Verilog-A files are writable). The newly-created ahdIShipDBs contain shared objects.

Anyone who references the same Verilog-A files can pick up the shared objects without setting any of the compiled C code environment variables and without needing write access to the directories containing the Verilog-A files.

## Turning Off Parallel Compilation

By default, parallel compilation is enabled when compiling the C code. You can turn off parallel compilation when the system load average is high. To turn off parallel compilation, set the `CDS_AHDL_COMPILE_C_MAX_LOAD` environment variable, as follows:

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

```
setenv CDS_AHDL_COMPILEC_MAX_LOAD max_load
```

**Note:** `max_load` is a user-specified float constant. When the load average is above `max_load`, parallel compilation is turned off. The load average is a measure of how many processes are on average concurrently demanding CPU attention. Usually, load average should be a float value between 0.8 and 2.5.

To turn on parallel compilation again, unset the `CDS_AHDL_COMPILEC_MAX_LOAD` environment variable as follows:

```
unsetenv CDS_AHDL_COMPILEC_MAX_LOAD
```

## Using Verilog-A Compact Models to Increase Simulation Speed

Verilog-A compact models are models of semiconductor devices used in analog simulators. The Verilog-A compiler treats modules that have a `compact_module` attribute as compact models and optimizes them accordingly to increase your simulation speed. For example:

```
(* compact_module *)  
module mosfet(drain, gate, source, bulk);
```

### *Important*

You must not use the `compact_module` attribute on modules that are not Verilog-A compact models.

**Note:** If you are using the Spectre solver, you can create *model cards*<sup>1</sup> for Verilog-A compact models.

See also

- [Noticing Differences When You Use the compact\\_module Attribute](#) on page 537
- [Specifying Instance and Model Parameters for a Verilog-A Compact Model](#) on page 537
- [Model Binning for Verilog-A Compact Models](#) on page 538
- [Compiling Verilog-A Compact Models for Reuse](#) on page 531
- [Reusing Verilog-A Shared Objects](#) on page 532

---

1. A *model card* is a Spectre language `model` statement (or `.model` for SPICE). For more information, see "Model Statements" in the "[Spectre Netlists](#)" chapter of the *Spectre Circuit Simulator User Guide*.



## Noticing Differences When You Use the `compact_module` Attribute

You might notice some of the following differences when you use the `compact_module` attribute:

- The software uses system calls in place of some arithmetic calculations. Spectre circuit simulator messages that result from these system calls (such as divide-by-zero) might appear different from messages that result from standard arithmetic calculations.
- Line number information is not available.
- The simulation result might differ slightly from a run where you do not use the `compact_module` attribute. The simulator shares some calculations to increase the simulation speed. The results are accurate enough for compact model applications.
- If you have a simulation that spends much of its time in DC analysis, you might be able to optimize the `initial_step` further by moving any I/O-related operations (such as `$strobe`) or signal-dependent (or bias-dependent) statements outside the `initial_step` or by removing them from the module entirely. For example:

```
@initial_step begin
    tmp =  $\bar{V}$ (in,out); // bias-dependent variable assignment
    ...
end
```

## Specifying Instance and Model Parameters for a Verilog-A Compact Model

You can use the `instance_parameter_list` attribute to distinguish between instance and model parameters for a Verilog-A compact model. If you do not use the `instance_parameter_list` attribute, the simulator interprets all parameters as instance parameters. When you use the `instance_parameter_list` attribute on a Verilog-A compact model, only those parameters you specify are instance and model parameters: All other parameters are model parameters. (Generally, you will have many more model parameters than instance parameters, so this mechanism lets you specify the smaller set of parameters, instance parameters, explicitly.)

The format of the `instance_parameter_list` attribute is as follows:

```
(* instance_parameter_list='{parameterList} *)
```

where `parameterList` is a comma-separated list of instance parameters. For example:

```
(* instance_parameter_list='{x,y} *)
(* compact_module *)
module mosfet(drain, gate, source, bulk);
```

You can specify the `instance_parameter_list` attribute only on a module you have marked as a Verilog-A compact model (using the `compact_module` attribute). You can specify the `instance_parameter_list` attribute either before or after the

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

`compact_module` attribute but they must occur together. The parameter list cannot contain string parameters (such as `'{"a", "b", "c"}`).

You can use the Spectre language `alter` and `altergroup` control statements with these instance/model parameters. (For information about `alter` and `altergroup`, see the "Control Statements" chapter of the *Spectre Circuit Simulator User Guide*.)

### Model Binning for Verilog-A Compact Models

Verilog-A compact models support model binning. To specify model binning for a Verilog-A compact model, you must add a `modelbin` attribute along with the `compact_module` attribute to the module. In addition, your module must have `wmax`, `wmin`, `lmax`, and `lmin` model parameters with `l` and `w` instance parameters that you declare using the `instance_parameter_list` attribute. For example:

```
(* compact_module *)
(* modelbin *)
(* instance_parameter_list='{x,y,l,w}' *) // l and w are binning parameters
module verilogAdefnName (drain, gate, source, bulk);
    parameter x=5;
    parameter y=10;
    parameter l=1.5u;
    parameter w=3u;
    ...
endmodule

model mos_model verilogAdefnName {
    1: ...lmin=0.5u lmax=1.5u wmin=1u wmax=2u
    2: ...lmin=1.5u lmax=2.5u wmin=2u wmax=3u
    3: ...lmin=2.5u lmax=2.5u ...
    4: ...
}

instanceName (nd ng ns nb) mos_model l=2u w=2u // 1.5u<=l<2.5u; 2u<=w<3u
```

The instance above falls into bin 2. To read more about binning, see "Binning" in the ["Parameter Specification and Modeling Feature"](#) chapter of the of the *Spectre Circuit Simulator User Guide*.

## Ignoring the State of a Verilog-A Module for RF Simulation

For Spectre RF simulation, use the `instrument_module` attribute immediately before the module declaration to designate a Verilog-A module as an instrument. The module must not be part of the actual circuit design (such that removing the module will not affect the circuit).

```
(* instrument_module *)
module ...(...);
...
endmodule
```

Instrumentation modules are Verilog-A modules that are either sources (modules that produce only outputs) or probes (modules that have only inputs). Instrumentation modules have hidden states. When you use the `instrument_module` attribute to flag a Verilog-A module as an instrument, Spectre RF ignores the state of the module and does not report it as a hidden state. Spectre RF effectively keeps the state of the module constant during RF analysis.

Verilog-A modules that produce only outputs include modulated sources such as CDMA, GSM, and DQPSK, as well as general circuit stimulus. Verilog-A modules that have only inputs include spectrum measurement modules (such as eye diagrams) and statistical correlations modules. Use the `instrument_module` attribute to designate these Verilog-A module as instruments for RF simulation. For example:

```
\include "constants.h"
\include "discipline.h"
(* instrument_module *)
module eye_diagram_generator(in_wave, x_axis, y_axis);
input in_wave;
...
endmodule
```

You can use the `instrument_module` attribute for Spectre RF PSS and envelope analyses, but not for Spectre RF QPSS analysis. For PSS analysis, the simulator saves the state of the Verilog-A module after the `tstab` iteration and keeps the state constant during the PSS shooting method iterations. For envelope analysis, the simulator evaluates the Verilog-A module state for each cycle and skips cycles only if the Verilog-A module state remains constant. If the Verilog-A state is changing, the simulator does not skip cycles and performs continuous cycles as long as the state is changing.

**Note:** For more information about PSS, envelope, and QPSS RF analyses, see the [\*Virtuoso Spectre Circuit Simulator RF Analysis User Guide\*](#).

## Ignoring the State of a Verilog-A Local Variable for RF Simulation

For Spectre RF simulation, use the `ignore_state` attribute immediately before a Verilog-A local variable declaration to designate it as a non-state variable. The variable must not be a state variable. For example:

```
(* ignore_state *)  
real nonStateVariable;
```

When you use the `ignore_state` attribute to indicate that a particular Verilog-A local variable is not a state variable, Spectre RF ignores the state of the variable (does not solve for a value) and does not report it as a hidden state. Spectre RF effectively keeps the state of the variable constant during RF analysis. For PSS analysis, the simulator saves the state of the Verilog-A local variable after the `tstab` iteration and keeps the state constant during the PSS shooting method iterations.

**Note:** For more information about Spectre RF analyses, see the [Virtuoso Spectre Circuit Simulator RF Analysis User Guide](#).

### Important

Using the `ignore_state` attribute can lead to incorrect results if you apply this attribute to any true-state variables (because Spectre RF ignores the state of the variable and does not solve for a value). If you have true-state variables that Spectre RF flags for hidden states, consider using the `instrument_module` attribute.

Consider the following example. Assume `vInMaxOld` and `vInMax` have earlier definitions in the Verilog-A block such that `vInMax=0.001` and `vInMaxOld=0.001`.

```
real vInMaxOld;  
real vInMax;  
analog begin  
vInMaxOld=vInMax; --- Hidden State  
if( V(in,hold) > 1.0 ) vInMax=V(in,hold);  
else vInMax=11.0*V(in,hold);  
V(out,hold) <+ (V(in,hold)-vInMax*V(n1,hold))/peakMag;  
....
```

At time  $t=0$ , the simulator has a value for `vInMax` (0.001). At the second time point, the simulator does not have a value for `vInMax` yet because it has not yet assigned a new value. Therefore, the simulator cannot assign a value to `vinMaxOld`, which causes a "hidden state" error. You can tag the `vinMaxOld` variable with the `ignore_state` attribute to prevent this hidden state error:

```
(* ignore_state *)  
real vInMaxOld;  
....
```

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

You can also use the `ignore_state` attribute to ignore variables that you use only to report debugging information. Because such variables do not affect the circuit state, you can use the `ignore_state` attribute safely to complete a simulation with accurate results.

**Note:** The `ignore_state` attribute is retired and should no longer be used. Cadence recommends that you use the `ignore_hidden_state` attribute instead.

## Ignoring the States of all variables in a Verilog-A Module for RF Simulation

For Spectre RF simulation, use the `ignore_hidden_state` attribute immediately before the module declaration to specify that a Verilog-A module does not contain any state variable. For example:

```
(* ignore_hidden_state *)  
Module ... (...);  
...  
endmodule
```

When you use the `ignore_hidden_state` attribute to indicate that a particular Verilog-A module does not contain any state variable, Spectre RF ignores the states of all variables in that module and does not report them as hidden states. Spectre RF effectively keeps the states of all variables in the module constant during RF analysis. For PSS analysis, the simulator saves the states of all the Verilog-A local variables after the `tstab` iteration and keeps the state constant during the PSS shooting method iterations.

For more information about Spectre RF analyses, see the *Virtuoso Spectre Circuit Simulator RF Analysis User Guide*.

### Important

Using the `ignore_hidden_state` attribute can lead to incorrect results if you apply this attribute to a module containing true-state variables (because Spectre RF ignores the states of all variables in the module and does not solve for their values). If you have a module that contains true-state variables that Spectre RF flags for hidden states, consider using the `instrument_module` attribute.

In the following example, assume that `vInMaxOld` and `vInMax` have earlier definitions in the Verilog-A block such that `vInMax=0.001` and `vInMaxOld=0.001`.

```
'include "constants.h"  
'include "discipline.h"  
module state(...);  
...  
analog begin  
vInMaxOld=vInMax; <-- Hidden State  
if( V(in,hold) > 1.0 ) vInMax=V(in,hold);  
else vInMax=11.0*V(in,hold);  
V(out,hold) <+ (V(in,hold)-vInMax*V(n1,hold))/peakMag;  
...  
endmodule
```

At time `t=0`, the simulator has a value for `vInMax` (`0.001`). At the second time point, the simulator does not have a value for `vInMax` yet because it has not yet assigned a new value. Therefore, the simulator cannot assign a value to `vInMaxOld`, which causes a "hidden state"

## Cadence Verilog-A Language Reference

### Getting Ready to Simulate

---

error. You can tag the state module with the `ignore_hidden_state` attribute to prevent this hidden state error:

```
'include "constants.h"
'include "discipline.h"
(* ignore_hidden_state *)
module state(...);
...
endmodule
```

If all variables in a Verilog-A module do not affect the circuit state, for example, all variables are used only to report debugging information, you can use the `ignore_hidden_state` attribute to complete a simulation with accurate results.

**Cadence Verilog-A Language Reference**  
Getting Ready to Simulate

---



---

## Supported and Unsupported Language Elements

---

The Cadence® Verilog®-A language is specified in Annex C of the *Verilog-AMS Language Reference Manual: Analog & Mixed-Signal Extensions to Verilog HDL*, produced by Open Verilog International.

The Cadence implementation of Verilog-A does not support the following elements of the specified Verilog-A language.

- Ordered parameter lists in hierarchical instantiation
- Hierarchical names, except for `node.potential.abstol` and `node.flow.abstol`, which are supported
- Derived natures
- The `generate` constructs
- String arrays
- The `defparam` statement
- Nested use of the `ddt` operator
- Vector branches
- Laplace transforms taking parameter-sized arrays as arguments
- The ``begin_keywords` and ``end_keywords` compiler directives
- Element reference among different named block

## Cadence Verilog-A Language Reference

### Supported and Unsupported Language Elements

---

The items in the next list are deprecated features. The Cadence implementation of Verilog-A supports these features, but might not in the future. These features are no longer supported in the standard specification of the language.

#### Deprecated features

---

Deprecated feature	To comply with the current standard,...
<code>`ifdef`</code>	Use <code>`ifdef</code> without a trailing tick. For example, instead of <code>`ifdef`CHECK_BACK_SURFACE</code>  <code>use</code> <code>`ifdef CHECK_BACK_SURFACE</code>
<code>`inf</code> used for specifications other than ranges	Use <code>`inf</code> only to specify ranges.
user-defined analog function	Use <code>analog function</code>
<code>discontinuity</code>	Use <code>\$discontinuity</code> .
<code>I(a, a)</code> to probe a port current	Use <code>I(&lt;a&gt;)</code> .
<code>delay</code>	Use <code>absdelay</code> .
Null statements used elsewhere other than in case and event statements	Use null statements (coded as <code>;</code> ) only in case or event control statements.
Chained assignment statements, such as <code>x=y=z</code>	Break the assignment chain into separate assignments, such as <code>y=z; x=y;</code> .
<code>\$limexp</code>	Use <code>limexp</code> .
Using <code>[]</code> for literal arrays	Use <code>{}</code> for literal arrays.
<code>bound_step</code>	<code>\$bound_step</code>
<code>export</code> qualifier	Delete the <code>export</code> qualifier, which is redundant.
<code>\$dist_</code> functions in the analog block	Use the corresponding <code>\$rdist_</code> function.
The second argument of the <code>cross</code> operator being a non-integer type	Change the second operator to an integer type.
Using <code>for</code> , <code>while</code> and <code>repeat</code> loop statements for the timer function	Use a <code>genvar</code> loop for the timer function.
Unassigned variables	Assign each variable. Unassigned variables are considered digital variables.

## Cadence Verilog-A Language Reference

### Supported and Unsupported Language Elements

---

#### Deprecated features

---

Deprecated feature	To comply with the current standard,...
<code>generate</code>	Use a genvar loop instead.
The second argument of the <code>last_crossing</code> operator being a non-integer type	Change the second operator to an integer type.

---

The items in the next list are Cadence extensions. These features are not part of the standard specification of the language.

#### Cadence extensions

---

##### Feature

---

Cadence syntax for attributes

`mfactor` attribute

`dynamicparams`

Inherited parameters

`$cdfs_set_rf_source_info`

`$cdfs_get_mc_trial_number`

---

**Cadence Verilog-A Language Reference**  
Supported and Unsupported Language Elements

---

---

## Creating ViewInfo for Verilog-A Cellview

---

This appendix describes a SKILL function that you can use to update the CDF information for a Verilog-A cellview. You might need to do this after copying a cellview.

### ahdlUpdateViewInfo

```
ahdlUpdateViewInfo( t_lib [?cell t1_cell [?view t1_view]] )
```

#### Description

Updates cellview CDF information. During the update, `ahdlUpdateViewInfo`: 1) parses the Verilog-A modules that define the specified cellviews; 2) issues any necessary error messages; 3) updates the cellview CDF information.

#### Arguments

<i>t_lib</i>	Name of the library to be updated.
<i>t1_cell</i>	Name or list of names of cells to be updated. If <i>t1_cell</i> is omitted, the function updates every veriloga cellview in the library.
<i>t1_view</i>	Name or list of names of cellviews to be updated. If <i>t1_view</i> is omitted, the function updates every veriloga cellview associated with the specified cell.

#### Example 1

```
ahdlUpdateViewInfo("myLibrary")
```

Updates all the veriloga cellviews in a library.

## **Example 2**

```
ahdlUpdateViewInfo("myLibrary" ?cell "res" "cmp" "opamp")
```

Updates three cells in a library.

## **Example 3**

```
ahdlUpdateViewInfo("myLibrary" ?cell "res" ?view "veriloga")
```

Updates one specified cellview.

---

## Converting SpectreHDL to Verilog-A

---

In general,

- If you are using ahdlLib (a 5x library), you need to change from using AHDL views to using Verilog-A views. There is a corresponding Verilog-A view for each AHDL view.
- If you are using a SpectreHDL file from *your\_install\_dir/tools/dfII/samples/artist/spectreHDL/SpectreHDL*, you need to use the corresponding Verilog-A file from *your\_install\_dir/tools/dfII/samples/artist/spectreHDL/Verilog-A* instead.

To convert a SpectreHDL file to a Verilog-A file, do the following:

### Important

For information about items you cannot convert, see [“SpectreHDL Constructs That Have No Verilog-A Equivalent”](#) on page 557.

1. Copy the SpectreHDL file to a name with a `.va` suffix. For example:

```
cp mySpectreHDLfile.def myVerilogAfile.va
```

2. Add the following lines to the top of your `.va` file:

```
\include "discipline.vams"  
\include "constants.vams"
```

3. Convert SpectreHDL constructs as appropriate based on the equivalents in the table in [“SpectreHDL Constructs That Have Verilog-A Equivalents”](#) on page 552.
4. Save and close the file.
5. Test your converted model.

## SpectreHDL Constructs That Have Verilog-A Equivalents

You can change the following SpectreHDL constructs to their Verilog-A equivalents as indicated:

Instead of...	Use...
<code>#define PARAM [value]</code>	<code>`define PARAM [value]</code>
<code>#include path/file</code>	<code>`include path/file</code>
<code>module (port_list) (param_list) { moddesc }</code>	<code>module (port_list); moddescr endmodule</code>
<code>input [V,I] name[, net2[...]] ; input [PotentialNature, FlowNature] net1[, net3];</code>	<code>input name[, net2[, net1, net3[,...]]; electrical name[, net2[, ...]]; discipline net1[, net3[, ...]];</code>  <b>Note:</b> Choose the appropriate discipline from the standard definitions file (installed in <code>your_install_dir/tools/spectre/etc/ahdl</code> ).
<code>node [V,I] portlist; node [V,I] internalNodelist;</code>	<code>inout portlist; electrical nodelist; electrical internalNodelist;</code>
<code>inout [V,I] nodelist;</code>	<code>inout nodelist; electrical nodelist;</code>
<code>output [V,I] nodelist;</code>	<code>output nodelist; electrical nodelist;</code>
<code>initial { if (analysis_list) {init_statements} } analog { statements } final { final_statements }</code>	<code>analog begin @(initial_step[(analysis_list)]) begin init_statements end statements @(final_step) begin final_statements end end</code>



## Cadence Verilog-A Language Reference

### Converting SpectreHDL to Verilog-A

#### Instead of...

#### Use...

<code>x++</code>	<code>x = x + 1;</code>
<code>x+=</code>	<code>x = x +;</code>
<code>x-=-</code>	<code>x = x -;</code>
<code>x*=-</code>	<code>x = x *;</code>
<code>x--</code>	<code>x = x - 1;</code>
<code>++a</code>	<code>a = a + 1;</code>
<code>--a</code>	<code>a = a - 1;</code>
<code>a += b</code>	<code>a = a + b;</code>
<code>a -= b</code>	<code>a = a - b;</code>
<code>a *= b</code>	<code>a = a * b;</code>
<code>a /= b</code>	<code>a = a / b;</code>
<code>PI</code>	<code>`M_PI</code>

**Note:** See also the set of supported constants in `your_install_dir/tools/spectre/etc/ahdl/constants.vams`.

For schematic blocks or Verilog-A definitions:

<code>blockname inst ( connectlist )</code> <code>(param1=value1,</code> <code>param2= value2);</code>	<code>blockname #(.param1(value1),</code> <code>.param2(value2)) inst</code> <code>(.port1(connect1),</code> <code>.port2(connect2));</code>
--	---

For primitives (defined in model files) or inline subcircuits:

<code>primname Inum (connectlist)</code> <code>(param1=val1, param2=val2);</code>	<code>primname #(.param1(value1),</code> <code>.param2(value2)) Inum</code> <code>(connect1, connect2);</code>
--	--

**Note:** You must match the connection order in the model or subcircuit definition.

<code>&lt;-</code>	<code>&lt;+</code>
enum type	Rewrite to use either a string or an integer instead of the enumerated type
stream	integer

**Note:** File pointers in Verilog-A are integers.

## Cadence Verilog-A Language Reference

### Converting SpectreHDL to Verilog-A

Instead of...	Use...
{ (the first one in the module)	<i>Remove it</i>
{ (except for the first one in the module)	begin
} (except for the last one in the module)	end
} (the last one in the module)	endmodule
\$transition	transition
\$threshold(...)	@ (cross(...))
<b>For example:</b>	<b>For example:</b>
\$threshold(V(vin1) - vtrans, 1)	@ (cross(V(vin1) - vtrans, 1))
\$break_point( <i>breakPoint</i> )	@ (timer( <i>time</i> ) )
<b>For example:</b>	<b>For example:</b>
if (\$break_point( nextBP )) {	@ (timer( nextBP )) begin
\$laplace_zp( <i>args</i> )	laplace_zp( <i>args</i> )
\$laplace_zd( <i>args</i> )	laplace_zd( <i>args</i> )
\$laplace_np( <i>args</i> )	laplace_np( <i>args</i> )
\$laplace_nd( <i>args</i> )	laplace_nd( <i>args</i> )
\$zi_zp( <i>args</i> )	zi_zp( <i>args</i> )
\$zi_zd( <i>args</i> )	zi_zd( <i>args</i> )
\$zi_np( <i>args</i> )	zi_np( <i>args</i> )
\$zi_nd( <i>args</i> )	zi_nd( <i>args</i> )
\$slew	slew
\$tdelay	absdelay
\$time	abstime
\$analysis( <i>arglist</i> )	analysis( <i>arglist</i> )
\$ac_stim	ac_stim
\$white_noise	white_noise

## Cadence Verilog-A Language Reference

### Converting SpectreHDL to Verilog-A

Instead of...	Use...
\$flicker_noise	flicker_noise
\$noise_table	noise_table
\$temp	\$temperature
dot	ddt
integ	idt
idtm0d	idtm0d
<pre>out = \$zdelay(expr, period, transtime, sampledelay, initvalue);</pre>	<pre>(if initvalue <i>specified</i>) @(initial_step) out1 = initvalue; (else) @(initial_step) out1 = expr; (endif) @(timer(sampledelay, period)) out1 = expr; out = transition(out1, 0, transtime, transtime);</pre>
\$halt	\$finish or \$stop
\$last_crossing	last_crossing
\$bound_step	\$bound_step
\$fread_table \$build_table \$interpolate	\$table_model
\$reltol	<i>node_name.nature.reltol</i>
\$abstol	<i>node_name.nature.abstol</i>
\$fread	\$fscanf
\$fwrite_table	\$strobe

## Cadence Verilog-A Language Reference

### Converting SpectreHDL to Verilog-A

---

<b>Instead of...</b>	<b>Use...</b>
<code>\$strcmp</code>	<code>== != &lt; &gt; &gt;= &lt;=</code>
<code>\$strlen</code>	<code>str.len</code>
<code>\$substr</code>	<code>str.substr(start,end)</code>
<code>\$strchr</code>	<code>str.getc</code>
<code>\$strcat</code>	<code>{str_des, str_src}</code>
<code>\$strtoreal</code>	<code>str.atoreal</code>
<code>\$strcpy</code>	<code>des_str = src_str</code>

## **SpectreHDL Constructs That Have No Verilog-A Equivalent**

The following SpectreHDL constructs have no Verilog-A equivalent:

- `$popen`
- `$pclose`
- `$system`
- `$ascii`
- `$strstr`

With SpectreHDL, you could define models inside the SpectreHDL language. For Verilog-A, you must move model definitions to separate files and include them in the Spectre netlist.

**Cadence Verilog-A Language Reference**  
Converting SpectreHDL to Verilog-A

---

---

## Verilog-A Source Protection

---

Cadence supports two different methods to protect (encrypt) source code, depending on what language you use. These two methods are similar but differ in the commands you use and in the implementation details.

Language	Method for Protection
Verilog-A	<code>ncprotect</code>
Spectre code	<code><u>spectre_encrypt</u></code>

When you use the `ncprotect` utility to prevent access to or modification of Verilog-A source code, you can

- Protect selected design units or models
- Protect selected regions within design units or models
- Automatically protect all design units and models in a file

Source protection prevents access to protected regions. When you use source protection, software or commands that normally report information that depends on code do not return any information that might reveal the contents of the protected regions. In addition, the simulator either suppresses warning and error messages from protected regions or issues generic messages that do not disclose protected information. You can use the protected code as usual in the simulation flow and it produces the same results as unprotected code.

**Note:** The `ncprotect` binary is available in the `<installation>/tools.<plat>/bin/` directory and not in the `<installation>/bin/` directory.

See the following topics for more information:

- [Protecting the Source Description of Selected Modules or Regions](#) on page 561
- [Using the Protection Pragmas](#) on page 562
- [The `ncprotect` Command](#) on page 563

## Cadence Verilog-A Language Reference

### Verilog-A Source Protection

---

- [Protecting All Modules in a Source Description](#) on page 565



## Protecting the Source Description of Selected Modules or Regions

To protect the source description of selected modules or regions,

1. Place protection pragmas in the source description to define the protected region.

The pragmas, which are in the form of comments, are

❑ `pragma protect`

Indicates the start of a protection block. Used in conjunction with `pragma protect begin`.

❑ `pragma protect begin`

Indicates the start of the data to be encrypted

❑ `pragma protect end`

Indicates the end of the data to be encrypted

For information about inserting the protection pragmas in your source code, see [“Using the Protection Pragmas”](#) on page 562.

2. Run the `ncprotect` command on the input files containing the regions to be protected.

This command creates a new source file in which the regions marked for protection are unreadable. By default, the new file has the same name as the original file, but with an appended `p`.

Ensure that the encrypted file is not changed after it is generated, perhaps by making the file read only. Changing the encrypted code by hand corrupts the file, causing error messages such as the following:

```
Error while decrypting : Corrupted encrypted block, checksum did not match
```

If you get such an error, you can resolve the problem by recreating or reinstalling the protected code.

To use the protected modules, you run the compiler as usual. The compiler decrypts the encrypted files and compiles the design units in the file. You can then elaborate the design and simulate the snapshot. Downstream programs provide restricted visibility and access to the protected units.

## Using the Protection Pragmas

You use the protection pragmas to mark regions for protection in Verilog-A code in your model files.

You can use the protection pragmas `protect begin` and `protect end` inside or outside of design units, provided that you pair each `protect begin` pragma with a `protect end` pragma in the same source file. If you insert a `protect begin` pragma without a corresponding `protect end` pragma, the software issues a warning and encrypts everything remaining in the file.

You can use multiple sets of the `protect begin` and `protect end` pragmas within design units. However, you cannot nest blocks of source code bounded by `protect begin` and `protect end` pragmas inside one another.

**Note:** The following tasks do nothing when they are located inside an that is protected: `$strobe`, `$fstrobe`, `$display`, `$fdisplay`, `$debug`, `$fdebug`, `$write`, `$fwrite`.

The following two examples show how to use the `protect begin` and `protect end` pragmas in a source file. The first example shows how to mark a region in the module `top_design` for protection:

```
module top_design (a, b, c)
    bottom_inst ();
// pragma protect
// pragma protect begin
    initial
        $display ("Inside module top_design");
// pragma protect end
endmodule
```

This next example shows how to mark an entire module, including the module name, for protection:

```
// pragma protect
// pragma protect begin
    module bottom ();
        initial
            begin
                $display ("Inside module bottom");
            end
        endmodule
// pragma protect end
```

## The nprotect Command

The `pragma protect`, `protect begin`, and `protect end` pragmas mark the regions you want to protect; encryption actually occurs when you run the `nprotect` command on the source description files. The syntax of the `nprotect` command is as follows:

```
nprotect [-options] hdl_source_file [hdl_source_file ...]
[-Append_log]
[-AUtoprotect]
[-Extension output_file_extension]
[-File filename]
[-Help]
[-Language {vlog | vhdl}]
[-Logfile logfile_name]
[-Messages]
[-NOCopyright]
[-NOLog]
[-NOStdout]
[-Overwrite]
[-Version]
```

For complete information, and many examples, see “`nprotect`” in the “Utilities” chapter of *NC-Verilog Simulator Help*.

Processing a source description with the `nprotect` command generally protects only the regions marked with `protect begin` and `protect end` pragmas. The command creates a new source file that differs from the original file in the following ways:

- The pragmas `protect begin` and `protect end` become `protect begin_protected` and `protect end_protected`, respectively. The software adds other pragmas for the encryption.
- The regions you marked for protection in the original source description become unreadable.

The protected version of the [first example](#) in the previous section takes the following form, allowing read access to the first two lines while encrypting the remainder of the module:

```
module top_design (a, b, c) // readable
    bottom_inst (); // readable
//pragma protect begin_protected
//pragma protect key_keyowner=Cadence Design Systems.
//pragma protect key_keyname=CDS_KEY
//pragma protect key_method=RC5
//pragma protect key_block
hjQ2rsuJMpL9F3O43Xx7zf656dz2xxBxdnHC0GvJFJG3Y5HL0dSoPcLMN5Zy6Iq+
ySMMWcOGkowbtoHVjNn3UdcZFD6NF1WHJpb7KIc8Php8iTluEZmtwTgDSy64yqLL
SCaqKffWXhnJ5n/936szbTSvc8vs2ILJYG4FnjIZeYARwKjbofvTgA==
//pragma protect end_key_block
//pragma protect digest_block
uilUH9+52Dwx1U6ajpWVBgZque4=
//pragma protect end_digest_block
//pragma protect data_block
jGzcQn3lBzXvF2kCXy+abmSjUdOfUzPOp7g7dfEzgN96O2ZRQP4aN7kqJOCA9shI
```

## Cadence Verilog-A Language Reference

### Verilog-A Source Protection

---

```
jcvO6pnBhjaTNlxUJBSbBA==  
//pragma protect end_data_block  
//pragma protect digest_block  
tzEpxTPg7KWB9yMYlqfoVE3lVk=  
//pragma protect end_digest_block  
//pragma protect end_protected  
endmodule
```

The new, protected source files do not overwrite the original, unprotected source files. When you protect the original source file with `ncprotect`, you can specify an optional file extension you want the software to append to the name of the protected source file. If you do not specify an extension, the `ncprotect` command automatically appends a `p` to the source file name to create the protected file name.

For example, the following command protects the file `src.v`. By default, the software appends a `p` to the protected source file name: `src.vp`.

```
ncprotect src.v
```

The following command specifies an extension `myext` for the protected version of `design.v`: `design.v.myext`.

```
ncprotect design.v -extension myext
```

**Note:** If the name of the protected file conflicts with the name of an existing file, the `ncprotect` command does not create the protected file; instead, it issues a message that alerts you to the conflict.

## Protecting All Modules in a Source Description

The `ncprotect -autoprotect` command (which you can use for Verilog-A code but not for Spectre code) protects all modules in the specified source file automatically. You do not need to insert the `protect begin` and `protect end` protection pragmas in any source description that you plan to compile with `-autoprotect`. If these pragmas already exist in your source file, the `ncprotect -autoprotect` command ignores them.

This option is particularly useful for protecting libraries that contain a large number of files with many modules.

**Cadence Verilog-A Language Reference**  
Verilog-A Source Protection

---

---

## Verilog-A Compliance

---

The Cadence® implementation of Verilog®-AMS and Verilog-A comply with the latest Verilog-AMS standard from Accellera: Accellera Verilog-AMS Version 2.3 (June 2009). The Verilog-A language is a subset of the Verilog-AMS language, but some of the language elements in the Verilog-A subset have changed since Verilog-A was released by itself (see the history outlined below). As a consequence, you might need to revise your Verilog-A modules before using them as Verilog-AMS modules.

**Note:** Accellera is the standards organization that defines the standard for Verilog-AMS and the Verilog-A subset.

History:

- OVI used to be the standards organization that defined the standard for Verilog-AMS and the Verilog-A subset and was incorporated into Accellera in the early 00's.
- Verilog-A was first standardized in OVI Verilog-A LRM Version 1.0 (August 1st, 1996).
- Verilog-AMS was first standardized in OVI Verilog-AMS LRM Version 2.0 (Feb 18th, 2000). In that LRM, Verilog-A was defined to be a subset of Verilog-AMS and certain backwardly-incompatible changes were made to the Verilog-A definition. In particular a number of usages were deprecated.

If your Verilog-A modules use any of the backwardly-incompatible changes made to the Verilog-A definition, you need to update your modules to be compliant.

**Note:** While the Cadence Verilog-A implementation continues to support many of these changes, we urge you to update your modules to avoid these usages because they are individually subject to removal in future releases. The software will issue warning messages when it encounters such usages; you should pay particular attention to these messages and update your modules accordingly.

## Making Your Models Compliant

To make your models compliant with the current standard, see the following topics as they apply to the language features you have used:

- [Analog Functions](#) on page 569
- [NULL Statements](#) on page 569
- [inf Used as a Number](#) on page 569
- [Changing Delay to Absdelay](#) on page 570
- [Changing \\$realtime to \\$abstime](#) on page 570
- [Changing bound\\_step to \\$bound\\_step](#) on page 570
- [Changing Array Specifications](#) on page 570
- [Chained Assignments Made Illegal](#) on page 571
- [Real Argument Not Supported as Direction Argument](#) on page 571
- [\\$limexp Changed to limexp](#) on page 571
- [`if `MACRO is Not Allowed](#) on page 572
- [discontinuity Changed to \\$discontinuity](#) on page 572



## Analog Functions

OVI Verilog-A 1.0 declaration of an analog function is

```
function name;
```

OVI Verilog-AMS 2.0 uses the syntax

```
analog function name;
```

**Suggested change:** Prefix all function declarations by the word `analog`. For example, change

```
function real foo;
```

to

```
analog function real foo;
```

**Verilog-A warning:** Yes

## NULL Statements

OVI Verilog-A 1.0 allows `NULL` statements to be used anywhere in an analog block. OVI Verilog-AMS 2.0 allows `NULL` statements to be used only after `case` statements or event control statements.

**Suggested change:**

Remove illegal `NULL` statements. For example, change

```
begin  
;  
end
```

to

```
begin  
end
```

**Verilog-A warning:** Yes

## inf Used as a Number

Spectre Verilog-A allows `inf` to be used as a number. OVI Verilog-AMS 2.0 allows `inf` to be used only on ranges.

**Suggested change:**

Change all illegal references to `inf` to a large number such as 1M. For example, change;

```
parameter real points_per_cycle = inf from [6:inf];
```

to

```
parameter real points_per_cycle = 1M from [6:inf];
```

**Verilog-A warning:** Yes

## Changing Delay to Absdelay

OVI Verilog-A 1.0 uses `delay` as the analog delay operator but OVI Verilog-AMS 2.0 uses `absdelay`.

**Suggested change:** Change `delay` to `absdelay`.

**Verilog-A error:** Yes

## Changing \$realtime to \$abstime

OVI Verilog-A 1.0 uses `$realtime` as absolute time but OVI Verilog-AMS 2.0 uses `$abstime`.

**Suggested change:** Change `$realtime` to `$abstime`.

**Verilog-A warning:** Yes

## Changing bound\_step to \$bound\_step

OVI Verilog-A 1.0 uses `bound_step` for step bounding but OVI Verilog-AMS 2.0 uses `$bound_step`.

**Suggested change:** Change `bound_step` to `$bound_step`.

**Verilog-A error:** Yes

## Changing Array Specifications

OVI Verilog-A 1.0 uses `[]` to specify arrays but OVI Verilog-AMS 2.0 uses `{}`.

**Suggested change:** Change `[]` to `{}`. For example, change

```
svcvcs #(.poles([-2*`PI*bw,0])) output_filter
```

to

```
svcvcs #(.poles({-2*`PI*bw,0})) output_filter
```

**Verilog-A warning:** Yes

## Chained Assignments Made Illegal

Spectre-Verilog-A allows chained assignments, such as  $x=y=z$ , but OVI Verilog-AMS 2.0 makes this illegal.

**Suggested change:** Break chain assignments into single assignments. For example, change

```
x=y=z;
```

to

```
y = z; x = y;
```

**Verilog-A warning:** Yes

## Real Argument Not Supported as Direction Argument

Spectre-Verilog-A allows real numbers to be used for the arguments of `@cross` and `last_crossing` but OVI Verilog-AMS 2.0 makes this illegal.

**Suggested change:** Change the real numbers to integers. For example, change

```
@(cross(V(in),1.0) begin
```

to

```
@(cross(V(in),1) begin
```

**Verilog-A warning:** Yes

## \$limexp Changed to limexp

OVI Verilog-A 1.0 uses `$limexp`, but OVI Verilog-AMS 2.0 uses `limexp`.

**Suggested change:** Change `$limexp` to `limexp`. For example, change

```
I(vp,vn) <+ is * ($limexp(vacross/$vt) - 1);
```

to

```
I(vp,vn) <+ is * (limexp(vacross/$vt) - 1);
```

**Verilog-A error:** Yes

### **`if `MACRO is Not Allowed**

Spectre-Verilog-A allows users to type ``if `MACRO`, but OVI Verilog-AMS 2.0, 1.0 and 1364 say this is illegal.

**Suggested change:** Change ``if `MACRO` to ``if MACRO` (Do not use the tick mark for the macro). For example, change

```
`ifdef `CHECK_BACK_SURFACE
```

to

```
`ifdef CHECK_BACK_SURFACE
```

**Verilog-A warning:** Yes

### **discontinuity Changed to \$discontinuity**

OVI Verilog-A 1.0 uses `discontinuity`, but OVI Verilog-AMS 2.0 uses `$discontinuity`.

**Suggested change:** Change `discontinuity` to `$discontinuity`.

**Verilog-A error:** Yes

## Noting Changes from OVI Verilog-AMS Version 2.0

The following table highlights changes between pre-OVI 2.0 and OVI 2.0 and beyond.

---

<b>Feature</b>	<b>Before OVI Version 2.0</b>	<b>OVI Version 2.0 and Beyond</b>
Empty discipline	Predefined as type <i>wire</i>	Type not defined
Implicit nodes	<code>`default_nodetype</code> <code><i>discipline_identifier</i></code> Default = <i>wire</i>	Default type: empty discipline, no domain type
<code>initial_step</code>	Default = TRAN	Default = ALL
<code>final_step</code>	Default = TRAN	Default = ALL
Discipline domain	N/A, assumed continuous	Continuous (default) and discrete

---

**Cadence Verilog-A Language Reference**  
Verilog-A Compliance

---

# Glossary

---

## A

### **analog HDL**

Also *AHDL*. An analog hardware description language for describing analog circuits and functions.

## B

### **behavioral description**

The 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 path between two nodes. Each branch has two associated quantities, a potential and a flow, with a reference direction for each.

## C

### **component**

The fundamental unit within a system. A component encapsulates behavior and structure. Modules and models can represent a single component, or a component with many subcomponents.

### **constitutive relationships**

The expressions and statements that relate the outputs, inputs, and parameters of a module. These relationships constitute a behavioral description.

**continuous context**

The context of statements that appear in the body of an analog block.

**control flow**

The conditional and iterative statements that control the behavior of a module. These statements evaluate variables (counters, flags, and tokens) to control the operation of different sections of a behavioral description.

**child module**

A module instantiated inside the behavioral description of another, “parent” module.

**D**

**declaration**

A definition of the properties of a variable, node, port, parameter, or net.

**discipline**

A user-defined binding of potential and flow natures and other attributes to a net. Disciplines are used to declare analog nets and can also be used as part of the declaration of digital nets.

**dynamic expression**

An expression whose value is derived from the evaluation of a derivative (the  $\frac{d}{dt}$  function). Dynamic expressions define time-dependent module behavior. Some functions cannot operate on dynamic expressions.

**E**

**element**

The fundamental unit within a system, which encapsulates behavior and structure (also known as a *component*).

**F**

**flow**

One of the two fundamental quantities used to simulate the behavior of a system. In electrical systems, flow is current.



## **G**

### **global declarations**

Declarations of variables and parameters at the beginning of a behavioral description.

### **ground**

The reference node, which has a potential of zero.

### **instance**

A named occurrence of a component created from a module definition. One module definition 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 an instance in a circuit or system.)

## **H**

### **hierarchical system**

A system in which the components are also systems.

## **K**

### **Kirchhoff's Laws**

Physical laws that define the interconnection relationships of nodes, branches, potentials, and flows. Kirchhoff's Laws specify a conservation of flow in and out of a node and a conservation of potential around a loop of branches.

## **L**

### **level**

One block within a behavioral description, delimited by a pair of matching keywords such as `begin-end`, `discipline-enddiscipline`.

### **leaf component**

A component that has no subcomponents.

## M

### **module**

A definition of the interfaces and behavior of a component.

### **model card**

A Spectre language `model` statement (or `.model` for SPICE). For more information, see "Model Statements" in the "[Spectre Netlists](#)" chapter of the *[Spectre Circuit Simulator User Guide](#)*.

## N

### **nature**

A named collection of attributes consisting of units, tolerances, and access function names.

### **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.

### **node**

A connection point of two or more branches in a graph. In an electrical system, and equipotential surface can be modeled as a node.

### **nondynamic expression**

An expression whose derivative with respect to time is zero for every point in time.

## P

### **parameter**

A variable used to characterize the behavior of an instance of a module. Parameters are defined in the first section of a module, the module interface declarations, and can be specified each time a module is instantiated.

### **parameter declaration**

The statement in a module definition that defines the instance parameters of the module.

### **port**

The physical connection of an expression in an instantiating (parent) module with an expression in an instantiated (child) module. A port of an instantiated module has two

## Cadence Verilog-A Language Reference

### Glossary

---

nets, the upper connection, which is a net in the instantiating module, and the lower connection, which is a net in the instantiated module.

#### **potential**

One of the two fundamental quantities used to simulate the behavior of a system. In electrical systems, potential is voltage.

#### **primitive**

A basic component that is defined entirely in terms of behavior, without reference to any other primitives.

#### **probe**

A branch introduced into a circuit (or system) that does not alter the circuit's behavior, but lets the simulator read the potential or flow at that point.

## **R**

#### **reference direction**

A convention for determining whether the flow through a branch, the potential across a branch, or the flow in or out of a terminal, is positive or negative.

#### **reference node**

The global node (which has a potential of zero) against which the potentials of all single nodes are measured. In an electrical system, the reference node is ground.

#### **run-time binding (of sources)**

The conditional introduction and removal of potential and flow sources during a simulation. A potential source can replace a flow source and vice versa.

## **S**

#### **scope**

The current nesting level of a block.

#### **seed**

A number used to initialize a random number generator, or a string used to initialize a list of automatically generated names, such as for a list of pins.

#### **signal**

1. A hierarchical collection of nets that, because of port connections, are contiguous.
2. A single valued function of time, such as voltage or current in a transient simulation.

**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.

**source**

A branch introduced between two nodes to contribute to the potential and flow of those nodes.

**system**

A collection of interconnected components that produces a response when acted upon by a stimulus.

**V**

**Verilog®-A**

A language for the behavioral description of continuous-time systems that uses a syntax similar to digital Verilog.

**Verilog-AMS**

A mixed-signal language for the behavioral description of continuous-time and discrete-time systems that uses a syntax similar to digital Verilog.

# Index

## Symbols

- ` (accent grave) [238](#)
- `define compiler directive [238](#)
  - modifying abstol with [527](#)
  - syntax [238](#)
  - tested by `ifdef compiler directive [240](#)
- `if `MACRO, not allowed [572](#)
- `ifdef compiler directive [240](#)
- `ifndef compiler directive [241](#)
- `include compiler directive [242](#)
- `resetall compiler directive [243](#)
- `timescale compiler directive [242](#)
- `undef compiler directive [240](#)
- ^ (bitwise binary exclusive OR) [98](#)
- ^~ (bitwise binary exclusive NOR) [98](#)
- \_ (underscore), in identifiers [49](#)
- ! (logical negation) [95](#)
- != (not equal to) [98](#)
- !=(inequality comparison string function) [103](#)
- (binary minus) [97](#)
- (unary minus) [95](#)
- ? and : (conditional operator) [100](#)
- .va file extension [520](#)
- " (double quote character), displaying [184](#)
- ( (left parenthesis) [62](#)
- (\* cds\_inherited\_parameter \*) [60](#)
- (\* compact\_module \*) [537, 538](#)
- (\* desc= ... \*) [57, 59, 75](#)
- (\* ignore\_state \*) [540](#)
- (\* inh\_conn\_prop\_name= ... \*) [226](#)
- (\* inherited\_mfactor \*) [228](#)
- (\* instance\_parameter\_list \*) [537, 538](#)
- (\* instrument\_module \*) [539](#)
- (\* modelbin \*) [538](#)
- (\* no\_rigid\_switch\_branch \*) [333](#)
- (\* units=... \*) [57, 59](#)
- ) (right parenthesis) [62](#)
- [ (left bracket), using to include end point in range [62](#)
- ] (right bracket), using to include end point in range [62](#)
- @ (at-sign) operator [118](#)
- \* (multiply) [97](#)
- / (divide) [97](#)
- /\* (slash, asterisk), as comment marker [48](#)
- // (double slash), as comment marker [48](#)
- \ (tab character), displaying [184](#)
- \ (backslash)
  - continuing macro text with [238](#)
  - displaying [184](#)
  - in escaped names [49](#)
- & (bitwise binary and) [98](#)
- && (logical and) [98](#)
- % (modulo) [97](#)
- % (percent character), displaying [184](#)
- + (binary plus) [96](#)
- + (unary plus) [95](#)
- < (less than) [97](#)
- <+ (branch contribution operator) [83](#)
- << (shift bits left) [98](#)
- <= (less than or equal) [97](#)
- == (logical equals) [97](#)
- > (greater than) [97](#)
- >= (greater than or equal) [97](#)
- >> (shift bits right) [98](#)
- | (bitwise binary or) [98](#)
- || (logical or) [98](#)
- ~ (bitwise unary negation) [95](#)
- ~^ (bitwise binary exclusive nor) [98](#)
- \$ (dollar sign), in identifiers [49](#)
- \$abstime function [131](#)
  - for RF analysis [131](#)
- \$display [187](#)
- \$display task [187, 188](#)
- \$dist\_chi\_square function [153](#)
- \$dist\_erlang function [154](#)
- \$dist\_exponential function [151](#)
- \$dist\_normal function [150](#)
- \$dist\_poisson function [152](#)
- \$dist\_t function [153](#)
- \$dist\_uniform function [149](#)
- \$fclose task [197](#)
- \$fdisplay [196](#)
- \$fdisplay task [196](#)
- \$fopen task [191](#)
  - special formatting commands for [192](#)
- \$fstrobe [195](#)
- \$fstrobe task [194, 195](#)
- \$fwrite [196](#)
- \$limexp

analog operator [161](#)  
     changed to limexp [571](#)  
 \$random simulator function [146](#)  
 \$realtime to \$abstime [570](#)  
 \$sscanf string function [104](#)  
 \$strobe [184](#)  
     description [184](#), [188](#)  
     example of use [186](#)  
 \$write [187](#), [188](#)

### A

above event [122](#)  
 abs function [112](#)  
 absolute function [112](#)  
 absolute paths [521](#)  
 absolute tolerances  
     modifying [527](#)  
     modifying in Cadence analog design environment [529](#)  
     modifying in standalone mode [527](#)  
     used to evaluate convergence [326](#)  
 absolute value [422](#)  
 absolute value model [422](#)  
 abstol  
     modifying in Cadence analog design environment [529](#)  
     modifying in standalone mode [527](#)  
 abstol attribute  
     in convergence [326](#)  
     description [69](#)  
     requirements for [70](#)  
 abstol in standalone mode  
     modifying [527](#)  
 abstol in the Cadence Analog Design Environment  
     modifying [529](#)  
 ac\_stim simulator function [143](#)  
 accent grave (`), compiler directive  
     designation [238](#)  
 access attribute  
     description [69](#)  
     requirements for [70](#)  
 access functions  
     name taken from discipline [137](#)  
     syntax [137](#)  
     using in branch contribution statement [83](#)  
     using to obtain values [138](#)  
     using to set values [138](#)

acos function [113](#)  
 acosh function [113](#)  
 ADC  
     8-bit differential nonlinearity measurement [439](#)  
     8-bit integral nonlinearity measurement [440](#)  
     definition [439](#)  
 ADC model  
     8-bit [466](#)  
     8-bit (ideal) [467](#)  
     8-bit differential nonlinearity measurement [439](#)  
     8-bit integral nonlinearity measurement [440](#)  
 Add Block form [255](#)  
 adder [423](#)  
 adder model [423](#)  
     four numbers [424](#)  
     full [387](#)  
     half [386](#)  
 adder, 4 numbers [424](#)  
 AHDL [575](#)  
 AHDL Linter [246](#)  
 ahdl variables, saving [291](#)  
 ahdl\_include statements  
     format [286](#)  
     syntax [524](#)  
 ahldomainerror option (Spectre) [113](#)  
 ahdlShipDB [533](#)  
 ahdlSimDB [533](#)  
 ahdlUpdateViewInfo [549](#)  
 ahdlUpdateViewInfo SKILL function [549](#)  
 AM demodulator [486](#)  
 AM demodulator model [486](#)  
 AM modulator [487](#)  
 AM modulator model [487](#)  
 ammeter (current meter) [441](#)  
 ammeter model [441](#)  
 amplifier [395](#)  
 amplifier model [395](#)  
     current deadband [340](#)  
     deadband differential [399](#)  
     differential [400](#)  
     limiting differential [407](#)  
     logarithmic [408](#)  
     operational [344](#)  
     sample-and-hold (ideal) [472](#)  
     variable gain differential [417](#)  
     voltage deadband [354](#)  
     voltage-controlled variable-gain [355](#)

- analog behavior, defining with control
  - flow [41](#)
- analog blocks
  - format example [40](#)
  - multiple blocks not allowed [40](#)
  - placement [40](#)
- analog components [339](#)
- analog events [117](#) to [124](#)
  - cross [120](#)
  - detecting [118](#)
  - detecting multiple [118](#)
  - final\_step [119](#)
  - initial\_step [119](#)
  - timer [124](#)
- analog functions [569](#)
- analog multiplexer [339](#)
- analog multiplexer model [339](#)
- analog operators [161](#)
  - \$limexp [161](#)
  - not allowed in for loop [89](#)
  - listed [161](#)
  - not allowed in repeat loop [88](#)
  - restrictions on [161](#)
  - using in looping constructs [90](#)
  - not allowed in while loop [89](#)
- analog systems [28](#)
- analog-to-digital converter
  - example [91](#)
  - model, 8-bit [466](#)
  - model, 8-bit (ideal) [467](#)
  - model, 8-bit differential nonlinearity
    - measurement [439](#)
  - model, 8-bit integral nonlinearity
    - measurement [440](#)
- analyses
  - detecting first time step in [119](#)
  - detecting last time step in [119](#)
- analysis function [141](#)
- analysis types [141](#)
- analysis-dependent functions [141](#)
- AND gate [372](#)
- AND gate model [372](#)
- angular velocity [316](#), [317](#)
- arc-cosine function [113](#)
- arc-hyperbolic cosine function [113](#)
- arc-hyperbolic sine function [113](#)
- arc-hyperbolic tangent function [113](#)
- arc-sine function [113](#)
- arc-tangent function [113](#)
- arc-tangent of x/y function [113](#)
- array specifications, changing [570](#)
- arrays
  - arguments represented as [173](#)
  - as parameter values [225](#)
  - assignment operator for [83](#)
  - of integers, declaring [56](#)
  - of parameters [62](#)
  - of reals, declaring [57](#)
- ASCII code, returning from character [102](#)
- asin function [113](#)
- asinh function [113](#)
- assignment operator, procedural [82](#)
- assignment statement, indirect branch [85](#)
- associated reference directions [29](#)
- association order, of operators [94](#)
- atan function [113](#)
- atan2 function [113](#)
- atanh function [113](#)
- atoi function
  - details [106](#)
- atoi operator [102](#)
- atoi string function [106](#)
- atoreal function [102](#)
  - details [106](#)
- atoreal string function [106](#)
- attenuator model [488](#)
- attributes
  - abstol [69](#)
  - access [69](#)
  - accessing [140](#)
  - blowup [70](#)
  - ddt\_nature [69](#)
  - huge [70](#)
  - idt\_nature [69](#)
  - requirements [70](#)
  - units [69](#)
  - user-defined [69](#)
  - using to define base nature [69](#)
- attributes (\*...\*)
  - cds\_inherited\_parameter [60](#)
  - compact\_module [537](#), [538](#)
  - desc [57](#), [59](#), [75](#)
  - ignore\_state [540](#)
  - inh\_conn\_prop\_name [226](#)
  - instance\_parameter\_list [537](#), [538](#)
  - instrument\_module [539](#)
  - interited\_mfactor [228](#)
  - modelbin [538](#)
  - no\_rigid\_switch\_branch [333](#)
  - units [57](#), [59](#)
- audio source [489](#)
- audio source model [489](#)

## B

base natures  
   declaring [69](#)  
   description [68](#)  
 basic components [356](#)  
 behavioral characteristics, defining with  
   internal nodes [44](#)  
 behavioral description, definition [575](#)  
 behavioral model, definition [575](#)  
 bidirectional ports [38](#)  
 binary operators [96](#)  
 binding, run-time, definition [579](#)  
 bit error rate calculator model [490](#)  
 bitwise operators [99](#)  
   AND [99](#)  
   exclusive NOR [99](#)  
   exclusive OR [99](#)  
   inclusive OR [99](#)  
   unary negation [100](#)  
 blanks, as white space [48](#)  
 block comment [48](#)  
 blocks  
   adding pins to [256](#)  
   adding to schematic [255, 256](#)  
   analog [40](#)  
   creating Verilog-A cellviews from [256](#)  
   definition [575](#)  
   freeform [256](#)  
   setting shape of [256](#)  
   using [255](#)  
 blowup attribute, description [70](#)  
 bound\_step simulator function [129](#)  
 bound\_step to \$bound\_step [570](#)  
 brackets ([ ]) [62](#)  
 branch contribution statement  
   compared with procedural assignment  
     statement [84](#)  
   cumulative effect of [84](#)  
   evaluation of [84](#)  
   incompatible with indirect branch  
     assignment [85](#)  
   syntax [83](#)  
 branch data type [77](#)  
 branch terminals [78](#)  
 branches  
   declaring [77](#)  
   definition [575](#)  
   flow, default value for [331](#)  
   port [328](#)

  reference directions for [29](#)  
   switch, creating [85](#)  
   switch, defined [331](#)  
   switch, equivalent circuit model for [331](#)  
   values associated with [29](#)  
 built-in primitives [324](#)  
 buses [75, 255](#)  
   supported by Verilog-A modules [255](#)

## C

c or C format character [185](#)  
 Cadence analog design environment  
   creating Verilog-A cellviews with [252](#)  
   netlister [286](#)  
   using multiple cellviews in [266](#)  
 Cadence analog design environment  
   Simulation window [291](#)  
 capacitor model [357](#)  
   untrimmed [351](#)  
 car  
   frame model [309](#)  
   on bumpy road, netlist for [314](#)  
   system model [313](#)  
 case construct [87](#)  
 case statement [87](#)  
 CDF parameter of view cyclic field [270](#)  
 CDF, definition [338](#)  
 CDS\_AHDLDMI\_ENABLE environment  
   variable [533](#)  
 CDS\_AHDLDMI\_SHIPDB\_COPY  
   environment variable [533, 534](#)  
 CDS\_AHDLDMI\_SHIPDB\_DIR environment  
   variable [533, 534](#)  
 CDS\_AHDLDMI\_SIMDB\_DIR environment  
   variable [533](#)  
 CDS\_MMSIM\_VERILOGA macro [523](#)  
 CDS\_VLOGA\_INCLUDE environment  
   variable [522](#)  
 cell bindings table [282](#)  
 Cellview From Cellview form [263](#)  
   using to create Verilog-A cellviews [257, 268](#)  
 cellviews  
   associating with instances [272, 282](#)  
   using Cadence analog design  
     environment to create [252](#)  
   changing default parameters of,  
     example [281](#)  
   creating by using the Cadence analog



## Cadence Verilog-A Language Reference

---

- design environment [252](#)
  - creating with Verilog-A editor [261](#)
  - deleting parameters from [271](#)
  - examining with Descend Edit [260](#)
  - names for [267](#)
  - overriding parameter defaults of [268](#)
  - switching [272](#)
    - example of [275](#)
    - type of, determined by software [267](#)
  - Cellviews Need Saving form [274](#)
  - chained assignments made illegal [571](#)
  - channel\_descriptor, returned by \$fopen [191](#)
  - characters
    - finding first instance in a string [107](#)
    - finding last instance in a string [108](#)
  - charge meter model [452](#)
  - charge pump model [491](#)
  - child modules
    - definition [576](#)
    - instantiating [285](#), [286](#)
  - chi-square distribution function [153](#)
  - circuit fault model
    - open [343](#)
    - short [346](#)
  - circular integrator operator
    - example [166](#)
    - using [164](#)
  - clamp model
    - hard current [341](#)
    - hard voltage [342](#)
    - soft current [347](#)
    - soft voltage [348](#)
  - clocked JK flip-flop model [380](#)
  - closing a file [197](#)
  - code generator model
    - 2-bit [492](#)
    - 4-bit [493](#)
  - comments [48](#)
    - in modules [48](#)
    - in text macros [239](#)
  - compact models [536](#)
    - model binning [538](#)
  - compact\_module [536](#)
  - comparator [396](#)
    - example [169](#)
    - model [396](#)
  - comparison operator
    - details [103](#)
  - comparison operators, for strings [102](#), [103](#)
  - compatibility
    - of disciplines [72](#)
    - node connection requirements [220](#)
    - of disciplines [72](#)
  - compensator model
    - lag [365](#)
    - lead [366](#)
    - lead-lag [367](#)
  - compilation, conditional [240](#)
  - compiled C code flow [531](#)
  - compiler directives
    - ``define` [238](#)
    - ``ifdef` [240](#)
    - ``ifndef` [241](#)
    - ``include` [242](#)
    - ``resetall` [243](#)
    - ``timescale` [242](#)
    - ``undef` [240](#)
  - designated by accent grave ( ` ) [238](#)
  - list of [238](#)
  - resetting to default values [243](#)
  - using [238](#)
- compiling code conditionally [240](#)
  - components
    - creating multiple cellviews for [267](#)
    - definition [575](#)
  - concatenation operator [103](#), [104](#)
    - details [103](#), [104](#)
  - concatenation operator, for strings [102](#)
  - conditional compilation [240](#)
  - conditional operator [100](#)
  - conditional statement [86](#)
  - configuration
    - needed for multiple cellviews [269](#)
    - opening in Cadence analog design environment [269](#), [275](#)
  - connecting instances
    - example [220](#)
    - rules for [220](#)
  - connecting the ports of module
    - instances [219](#)
  - conservative discipline [72](#)
  - conservative systems [29](#)
    - conservative disciplines used to define [77](#)
    - defined [29](#)
    - values associated with [29](#)
  - constant expression [94](#)
  - constant power sink model [345](#)
  - constants
    - integer [50](#)
    - real [52](#)

string, used as parameters [226](#)  
 constants.vams file  
   location of [521](#), [522](#)  
   role in simulation [521](#)  
 constitutive equations [319](#)  
 constitutive relationships  
   definition [324](#), [575](#)  
   use in nodal analysis [325](#)  
 constructs  
   case [87](#)  
   looping [90](#)  
   procedural control [81](#)  
 contribution statements, format [40](#), [83](#)  
 control components [364](#)  
 control flow  
   definition [576](#)  
   describing behavior with [41](#)  
 controlled integrator model [397](#)  
 controlled sources [330](#)  
 controller model  
   proportional [368](#)  
   proportional derivative [369](#)  
   proportional integral [370](#)  
   proportional integral derivative [371](#)  
 convergence [325](#)  
 conversion specifications [185](#)  
 converting real numbers to integers [57](#)  
 copy operator, for strings [102](#)  
 core model, magnetic [418](#)  
 cos function [113](#)  
 cosh function [113](#)  
 cosine function [113](#)  
 Create New File form [261](#)  
 cross event [120](#)  
 cross function  
   syntax [120](#)  
 cube model [425](#)  
 cubic root model [426](#)  
 current analysis type, determining [141](#)  
 current clamp model  
   hard [341](#)  
   soft [347](#)  
 current deadband amplifier model [340](#)  
 current meter model [441](#)  
 current source model  
   current-controlled [362](#)  
   voltage-controlled [361](#)  
 current-controlled current source [331](#), [362](#)  
 current-controlled current source  
   model [362](#)  
 current-controlled voltage source [330](#), [360](#)

current-controlled voltage source  
   model [360](#)

## D

d or D format character [185](#)  
 DAC model  
   8-bit [469](#)  
   8-bit (ideal) [470](#)  
   8-bit differential nonlinearity  
     measurement [442](#)  
   8-bit integral nonlinearity  
     measurement [443](#)  
 DAC, definition [442](#)  
 damper model [460](#)  
 data types  
   branch [77](#)  
   discipline [71](#)  
   integer number [56](#)  
   nature [68](#)  
   parameter [58](#)  
   real number [57](#)  
 DC analysis  
   value returned by idt during [163](#)  
 DC motor model [392](#)  
 ddt operator (time derivative) [43](#), [162](#)  
 ddt\_nature attribute  
   description [69](#)  
   requirements for [70](#)  
 deadband amplifier model  
   current [340](#)  
   voltage [354](#)  
 deadband differential amplifier model [399](#)  
 deadband model [398](#)  
 decider model [494](#)  
 decimal logarithm function [112](#)  
 decimator model [468](#)  
 declarations  
   definition [576](#)  
   global, definition [577](#)  
   .def filename extension [337](#)  
 default values, required for parameters [61](#)  
 `define compiler directive  
   modifying abstol with [527](#)  
   syntax [238](#)  
 delay operator [167](#)  
 delay to absdelay [570](#)  
 delaying continuously valued  
   waveform [167](#)  
 deleting parameters from a veriloga or ahdl

## Cadence Verilog-A Language Reference

---

- Cellview [271](#)
- delta probe model [444](#)
- demodulator model
  - 8-bit PCM [502](#)
  - AM [486](#)
  - FM [497](#)
  - PM [506](#)
  - QAM 16-ary [508](#)
  - QPSK [511](#)
- derivative controller model
  - proportional [369](#)
  - proportional integral [371](#)
- derivative, time [162](#)
- derived nature [68](#)
- descend dialog [277](#)
- descend edit [260](#)
- descend edit command [260](#)
- describing a system [27](#)
- description attribute
  - for integers [57](#)
  - for net disciplines [75](#)
  - for parameter declarations [59](#)
  - for reals [57](#)
- differential amplifier (opamp) [400](#)
- differential amplifier model [400](#)
  - deadband [399](#)
  - limiting [407](#)
  - variable gain [417](#)
- differential signal driver [401](#)
- differential signal driver model [401](#)
- differentiator model [402](#)
- digital phase locked loop model [495](#)
- digital to analog converter example [172](#)
- digital voltage controlled oscillator model [496](#)
- digital-to-analog converter model
  - 8-bit [469](#)
  - 8-bit (ideal) [470](#)
  - 8-bit differential nonlinearity measurement [442](#)
  - 8-bit integral nonlinearity measurement [443](#)
- diode model [478](#)
  - Schottky [485](#)
- direction of ports, declaring [37](#)
- directions, reference [579](#)
- directives. *See* compiler directives
- disciplines [71](#)
  - compatibility of [72](#) to [75](#)
  - conservative [72](#)
  - declaring [71](#)
  - definition [576](#)
  - empty [72](#)
  - empty, declaring terminals with [76](#)
  - scope of [71](#)
  - signal-flow [72](#)
- disciplines.vams file
  - location of [521](#), [522](#)
  - required in Cadence analog design environment [529](#)
  - role in simulation [521](#)
- discontinuities
  - announcing [127](#)
  - in switch branches [332](#)
- discontinuity function
  - changed to \$discontinuity [572](#)
  - not required for switch branches [332](#)
  - syntax [127](#)
- discrete-time finite difference approximation [325](#)
- \$display task [187](#), [188](#)
- displaying
  - information [183](#)
  - results [183](#)
  - waveforms of variables [291](#)
- \$dist\_chi\_square function [153](#)
- \$dist\_erlang function [154](#)
- \$dist\_exponential function [151](#)
- \$dist\_normal function [150](#)
- \$dist\_poisson function [152](#)
- \$dist\_t function [153](#)
- \$dist\_uniform function [149](#)
- distributions
  - chi-square [153](#)
  - Erlang [154](#)
  - exponential [151](#)
  - gaussian [151](#)
  - normal [150](#)
  - Poisson [152](#)
  - Student's T [153](#)
  - uniform [149](#)
- divider model [427](#)
- DNL, definition [439](#)
- dollar signs, in identifiers [49](#)
- domain
  - of hyperbolic functions [113](#)
  - of mathematical functions [112](#)
  - of trigonometric functions [113](#)
- driver model
  - differential signal [401](#)
- D-type flip-flop model [379](#)
- dynamic expression, definition [576](#)

## E

e or E format character [185](#)  
 Edit Object Properties form [278](#)  
 8-bit parallel register model [390](#)  
 8-bit serial register model [391](#)  
 electrical modeling [296](#)  
 electromagnetic components [392](#)  
 electromagnetic relay [393](#)  
 electromagnetic relay model [393](#)  
 element, definition [576](#)  
 else statement, matching with if statement [87](#)  
 empty disciplines  
     compatibility of [73](#)  
     definition [72](#)  
     example [72](#)  
     predefined (wire) [72](#)  
 endmodule keyword [34](#)  
 enumerated values, as parameter values [226](#)  
 environment functions [131](#)  
 environment variables  
     CDS\_AHDLDMI\_ENABLE [533](#)  
     CDS\_AHDLDMI\_SHIPDB\_COPY [533](#),  
         [534](#)  
     CDS\_AHDLDMI\_SHIPDB\_DIR [533](#),  
         [534](#)  
     CDS\_AHDLDMI\_SIMDB\_DIR [533](#)  
     CDS\_VLOGA\_INCLUDE [522](#)  
 equality comparison string function  
     (==) [103](#)  
 Erlang distribution [154](#)  
 Erlang distribution function [154](#)  
 error calculation block [364](#)  
 error calculation block model [364](#)  
 error messages, forms of [517](#)  
 escaped names [49](#)  
     defined [49](#)  
     in Cadence analog design environment [264](#)  
     Spectre [49](#)  
     using in the Cadence analog design environment [264](#)  
 event OR operator [118](#)  
 events  
     detecting analog [118](#)  
     detecting and using [118](#)  
 events, analog [117](#) to [124](#)  
 examples

\$strobe formatting [186](#)  
 analog-to-digital converter [91](#)  
 car [314](#)  
 gearbox [319](#)  
 ideal relay [332](#)  
 ideal sampled data integrator [183](#)  
 inductor [42](#)  
 limiter [310](#)  
 linear damper [309](#)  
 motor [300](#)  
 rectifier [296](#)  
 RLC circuit [44](#)  
 road [310](#)  
 shock absorber [309](#)  
 sources and probes [334](#)  
 spring [308](#)  
 thin-film transistor [301](#)  
 thyristors [297](#)  
 transformer [298](#)  
 voltage deadband amplifier [41](#)  
 wheel [312](#)  
 exclude keyword [62](#)  
 exp function [112](#)  
 exponential distribution function [151](#)  
 exponential function [112](#)  
 exponential function model [428](#)  
 exponential function, limited [161](#)  
 expressions  
     constant [94](#)  
     definition [94](#)  
     dynamic, definition [576](#)  
     short-circuiting of [101](#)

## F

f or F format character [185](#)  
 fault model  
     open circuit [343](#)  
     short circuit [346](#)  
 \$fclose task [197](#)  
 \$fdisplay task [196](#)  
 file extension .va [520](#)  
 files  
     closing [197](#)  
     including at compilation time [242](#)  
     opening [191](#)  
     writing to [195](#)  
 files, working with [191](#)  
 filters  
     slew [171](#)

transition [168](#)  
 final\_step event [119](#)  
 find event probe [445](#)  
 find event probe model [445](#)  
 find slope [447](#)  
 find slope model [447](#)  
 finite-difference approximation [325](#)  
 flicker\_noise function [144](#)  
 flicker\_noise simulator function [144](#)  
 flip-flop model  
   clocked JK [380](#)  
   D-type [379](#)  
   JK-type [382](#)  
   RS-type [384](#)  
   toggle-type [385](#)  
   trigger-type [385](#)  
 flow  
   default value for [331](#)  
   definition [576](#)  
   in a conservative system [29](#)  
   probes, definition [328](#)  
   probes, in port branches [329](#)  
   sources, definition [329](#)  
   sources, equivalent circuit model  
     for [330](#)  
   sources, switching to potential  
     sources [331](#)  
 flow law. *See* Kirchhoff's Laws, Flow Law  
 flow-to-value converter model [403](#)  
 FM demodulator [497](#)  
 FM demodulator model [497](#)  
 FM modulator model [498](#)  
 \$fopen task [191](#)  
 for loop statement [89](#)  
 for statement [89](#)  
 formatting output [185](#)  
 forms  
   Add Block [255](#)  
   Cellview From Cellview [257](#), [263](#), [268](#)  
   Cellviews Need Saving [274](#)  
   Create New File [261](#)  
   Edit Object Properties [278](#)  
   New Library [253](#)  
   Open Configuration or Top  
     Cellview [269](#)  
   Simulation Environment Options [289](#)  
   Symbol Generation [262](#)  
   Technology File for New Library [254](#)  
 four-number adder model [424](#)  
 four-number subtractor model [438](#)  
 freeform block shape [256](#)

frequency meter model [448](#)  
 frequency-phase detector model [499](#)  
 \$fstrobe task [194](#), [195](#)  
 full adder model [387](#)  
 full subtractor model [389](#)  
 full wave rectifier model, two phase [475](#)  
 functional blocks [395](#)  
 functions  
   access [137](#)  
   environment [131](#)  
   mathematical [111](#)  
   string [101](#)  
   user-defined [202](#)

## G

g or G format character [185](#)  
 gain block [214](#)  
 gap model, magnetic [419](#)  
 gate pulses, used to control thyristors [297](#)  
 gaussian distribution [151](#)  
 gearbox  
   behavioral description for [319](#)  
   model [315](#)  
   netlist for [320](#)  
 gearbox model [459](#)  
 generate statement [90](#)  
 generating random numbers [146](#)  
 generating random numbers in specified  
   distributions [149](#)  
 genvars [66](#)  
 getc function [102](#)  
 getc string function [107](#)  
 global declarations, definition [577](#)  
 ground nodes  
   as assumed branch terminal [78](#)  
   compatibility of [220](#)  
   potential of [29](#)

## H

h or H format character [185](#)  
 half adder model [386](#)  
 half subtractor model [388](#)  
 half wave rectifier model, two phase [476](#)  
 hard current clamp model [341](#)  
 hard voltage clamp model [342](#)  
 hierarchical module instantiation [286](#)  
 hierarchical name, displaying [184](#)

hierarchical Verilog-A modules [286](#)  
Hierarchy Editor  
    synchronizing with schematic [274](#)  
    window [272](#)  
hierarchy, using [288](#)  
higher order systems [44](#)  
huge attribute, description [70](#)  
hyperbolic cosine function [113](#)  
hyperbolic functions [113](#)  
hyperbolic sine function [113](#)  
hyperbolic tangent function [113](#)  
hypot function [113](#)  
hypotenuse function [113](#)  
hysteresis model, rectangular [404](#)

## I

IC analysis, value returned by idt  
    during [163](#)  
ideal relay example [332](#)  
ideal sampled data integrator example [183](#)  
identifiers [48](#)  
idt operator  
    example [43](#)  
    using in feedback configuration [164](#)  
idt\_nature attribute  
    description [69](#)  
    requirements for [70](#)  
idtmod operator  
    example [166](#)  
    using [164](#)  
`ifdef compiler directive [240](#)  
`ifndef compiler directive [241](#)  
ignored code, restrictions on [241](#)  
impedance meter model [458](#)  
implicit branches [78](#)  
implicit models [335](#)  
include Compiler Directive [520](#)  
`include compiler directive [242](#)  
including files  
    at compilation time [242](#)  
    in a netlist [524](#)  
including Verilog-A through model  
    setup [286](#)  
indirect branch assignment statement [85](#)  
inductor model [358](#)  
    module describing [42](#)  
    untrimmed [352](#)  
inequality comparison string function  
    (!=) [103](#)  
inertia [317](#)  
-inf (negative infinity) [62](#)  
inf used as a number [569](#)  
infinity, indicating in a range [62](#)  
inh\_conn\_def\_value attribute [226](#)  
inh\_conn\_prop\_name attribute [226](#)  
inherited connections, attributes for [226](#)  
inherited parameters, attribute for [60](#)  
inherited ports, using [226](#)  
inherited\_mfactor attribute [228](#)  
initial\_step event [119](#)  
    example of use [310](#)  
    syntax [119](#)  
instance  
    statement. See module instance  
        statement example  
instance parameters, modifying [281](#)  
instances  
    associating cellviews with [272](#)  
    connecting with ports [219](#), [220](#)  
    creating [214](#)  
    creating and naming [214](#)  
    definition [577](#)  
    examining Verilog-A modules bound  
        to [276](#)  
    labels for, in Cadence analog design  
        environment [256](#)  
    overriding parameter values in [220](#)  
instantiating  
    analog primitives [223](#)  
    analog primitives that use array valued  
        parameters [225](#)  
    module description files in netlists [523](#)  
    modules that use unsupported parameter  
        types [226](#)  
    modules with netlists [45](#)  
    Verilog-A modules [214](#)  
instantiation  
    definition [577](#)  
    hierarchical [523](#)  
    of non-Verilog-A modules [226](#)  
    syntax [214](#)  
instrumentation module [539](#)  
integer  
    attributes for [57](#), [333](#)  
    constants [50](#)  
    data type [56](#)  
    declaring [56](#)  
    numbers [50](#), [56](#)  
    range allowed in Verilog-A [56](#)  
integral controller model, proportional [370](#)

integral derivative controller model,  
    proportional [371](#)  
integral, time [162](#)  
integration and differentiation with analog  
    signal, using [42](#)  
integrator [405](#)  
integrator model [405](#)  
    controlled [397](#)  
    saturating [412](#)  
    switched capacitor [474](#)  
interconnection relationships [324](#)  
interface declarations, example [36](#)  
internal nodes  
    for higher order derivatives [42](#)  
    in higher order systems [44](#)  
    use [43](#)  
internal nodes in behavioral definitions,  
    using [44](#)  
internal nodes in higher order system,  
    using [44](#)  
internal nodes in modules, using [43](#)  
interpolating with table models [156](#)

## J

JK-type flip-flop model [382](#)

## K

Kirchhoff's Laws [324](#)  
    definition [577](#)  
    Flow Law [29](#), [324](#), [325](#), [326](#)  
    illustrated [324](#)  
    use in nodal analysis [325](#)  
    Potential Law [29](#), [324](#)

## L

lag compensator model [365](#)  
Laplace transforms  
    numerator-denominator form [176](#)  
    numerator-pole form [175](#)  
    s-domain filters [173](#)  
    zero-denominator form [175](#)  
    zero-pole form [174](#)  
laplace\_nd Laplace transform [176](#)  
laplace\_np Laplace transform [175](#)  
laplace\_zd Laplace transform [175](#)

laplace\_zp Laplace transform [174](#)  
last\_crossing simulator function  
    improving accuracy of [130](#)  
    setting direction for [120](#), [130](#)  
    syntax [130](#)  
laws, Kirchhoff's. *See* Kirchhoff's Laws  
lead compensator model [366](#)  
lead-lag compensator model [367](#)  
left justifying output [185](#)  
len function [102](#)  
len string function [107](#)  
level shifter model [383](#), [406](#)  
level, definition [577](#)  
libraries  
    creating [252](#)  
\$limexp analog operator [161](#)  
limited exponential function [161](#)  
limiter model [310](#)  
limiting differential amplifier model [407](#)  
linear conductor model [334](#)  
linear damper model [309](#)  
linear resistor model [335](#)  
ln function [112](#)  
local parameters  
    declaring [58](#)  
log function [112](#)  
logarithm function  
    decimal [112](#)  
    natural [112](#)  
logarithmic amplifier model [408](#)  
logic components [372](#)  
logic table [382](#), [384](#), [385](#)  
lowercase characters, required for SPICE-  
    mode netlisting [526](#)  
LPF, definition [495](#)

## M

m factor (multiplicity factor)  
    attributes for [228](#)  
    example using [228](#)  
    using [228](#)  
.m suffix, required for models [291](#)  
macros  
    CDS\_MMSIM\_VERILOGA [523](#)  
macros. *See* text macros  
magnetic components [418](#)  
magnetic core [418](#)  
magnetic core model [418](#)  
magnetic gap [419](#)

magnetic gap model [419](#)  
 magnetic winding [420](#)  
 magnetic winding model [420](#)  
 mapping instance ports to module ports [215](#)  
 mapping ports by name [216](#)  
 mapping ports by order [215](#)  
 mass model [461](#)  
 math domain errors, controlling [113](#)  
 mathematical components [422](#)  
 mathematical functions [112](#)  
 maximum (max) function [112](#)  
 measure components [439](#)  
 measurement model  
     offset [449](#)  
     slew rate [449, 454](#)  
 mechanical damper [460](#)  
 mechanical damper model [460](#)  
 mechanical mass [461](#)  
 mechanical mass model [461](#)  
 mechanical modeling [307](#)  
 mechanical restrainer [462](#)  
 mechanical restrainer model [462](#)  
 mechanical spring [464](#)  
 mechanical spring model [464](#)  
 mechanical systems [459](#)  
 messages, error [517](#)  
 minimum (min) function [112](#)  
 mixed conservative and signal-flow systems [29](#)  
 mixed-signal components [466](#)  
 mixer [500](#)  
 mixer model [499, 500](#)  
 model binning for compact models [538](#)  
 model file [291](#)  
 modeling [297](#)  
 models  
     in modules [290](#)  
     library of samples [337](#)  
     using in a Verilog-A module [290](#)  
     using with a Verilog-A [290](#)  
 modulator model  
     8-bit PCM [503](#)  
     AM [487](#)  
     FM [498](#)  
     PM [507](#)  
     QPSK [512](#)  
     quadrature amplitude 16-ary [510](#)  
 module keyword [34](#)  
 modules  
     analog behavior of

        defining [39](#)  
         behavioral description [39](#)  
         capacitor example [42](#)  
         child, definition [576](#)  
         declaring [34](#)  
         definition [34, 578](#)  
         format [34](#)  
         format example [34](#)  
         hierarchy of [213](#)  
         instance statement, example [215, 220](#)  
         instantiating in other modules [214](#)  
         interface declarations [36](#)  
         interface, declaring [36](#)  
         internal nodes in [43](#)  
         name [36](#)  
         netlist instantiation of [45](#)  
         using nodes in [76](#)  
         non-Verilog-A [226](#)  
         overview [34](#)  
         RLC circuit example [44, 45](#)  
         top-level [213](#)  
         transformer example [214](#)  
         voltage deadband amplifier example [41](#)  
 MOS thin-film transistor [481](#)  
 MOS thin-film transistor model [481](#)  
 MOS transistor (level 1) [479](#)  
 MOS transistor model (level 1) [479](#)  
 motor model  
     behavioral description for [300](#)  
     DC [392](#)  
     three-phase [394](#)  
 multilevel hierarchical designs [285](#)  
 multiple cellviews, using for instances [266](#)  
 multiplexer model [409](#)  
 multiplier model [429](#)

## N

N JFET transistor model [482](#)  
 named branches [77](#)  
 names, escaped [49](#)  
 naming requirements for SPICE-mode netlisting [526](#)  
 NAND Gate [373](#)  
 NAND gate model [373](#)  
 natural log function model [430](#)  
 natural logarithm function [112](#)  
 natures [68](#)  
     access function for [69](#)  
     attributes [69](#)



base, declaring [69](#)  
 base, definition [68](#)  
 binding with potential and flow [71](#)  
   declaring [68](#)  
   definition [578](#)  
   deriving from other natures [68](#)  
   requirements for [68](#)  
 ncprotect command [563](#)  
 ncprotect, using [559](#)  
 net disciplines [75](#)  
   description attribute for [75](#)  
 netlisting Verilog-A modules [286](#)  
 netlists  
   creating [523](#)  
   example using cellviews [279](#)  
   including files in [524](#)  
   including Verilog-A modules in [286](#)  
   instantiating module description files  
     in [45](#)  
   n-channel TFT device [306](#)  
   preparing to display waveforms [291](#)  
   VCO2 example [288](#)  
 New Library form [253](#)  
 new-line characters  
   as white space [48](#)  
   displaying [184](#)  
 Newton-Raphson method  
   definition [578](#)  
   used to evaluate systems [325](#)  
 no\_rigid\_switch\_branch attribute [333](#)  
 nodal analysis [325](#)  
 node data type [75](#)  
 nodes [28](#)  
   assumed to be infinitely small [324](#)  
   connecting instances with [219](#)  
   declaring [75](#)  
   definition [578](#)  
   matching sizes required when  
     connected [220](#)  
   as module ports [76](#)  
   reference, definition [579](#)  
   reference, potential of [29](#)  
   scalar [75](#)  
   values associated with [29](#)  
   vector, declaring [75](#)  
   vector, definition [75](#)  
   ways of using [76](#)  
 noise functions  
   flicker\_noise [144](#)  
   noise\_table [145](#)  
 noise source model [501](#)

noise\_table function [145](#)  
 noise\_table simulator function [145](#)  
 NOR Gate [376](#)  
 NOR gate model [376](#)  
 normal (gaussian) distribution [150](#)  
 normal distribution function [150](#)  
 NOT Gate [375](#)  
 NOT gate model [375](#)  
 NPN bipolar junction transistor model [483](#)  
 NR method, definition [578](#)  
 NULL statements [569](#)  
 numbers [50](#)  
 numerator-denominator Laplace  
   transforms [176](#)  
 numerator-denominator Z-transforms [182](#)  
 numerator-pole Laplace transforms [175](#)  
 numerator-pole Z-transforms [181](#)

## O

o or O format character [186](#)  
 offset measurement [449](#)  
 offset measurement model [449](#)  
 one-line comment [48](#)  
 opamp model [344, 400](#)  
 open circuit fault [343](#)  
 open circuit fault model [343](#)  
 Open Configuration or Top Cellview  
   form [269](#)  
 opening  
   design [275](#)  
   file [191](#)  
 opening a configuration and associated  
   schematic [269](#)  
 operation [296](#)  
 operational amplifier model [344](#)  
 operators [93 to 100](#)  
   analog [161](#)  
   association of [94](#)  
   binary [96](#)  
   bitwise [99](#)  
   circular integrator [164](#)  
   delay [167](#)  
   idtmod [164](#)  
   precedence [101](#)  
   precedence of [94, 101](#)  
   string [101](#)  
   ternary [100](#)  
   time derivative [162](#)  
   time integral [162](#)

- unary [95](#)
- options
  - ahdldomainerror (Spectre) [113](#)
- or (event OR) [98](#)
- OR Gate [374](#)
- OR gate model [374](#)
- OR operator, event [118](#)
- order of evaluation, changing [94](#)
- ordered lists, mapping nodes with [215](#)
- ordinary identifiers [49](#)
- oscillator model
  - digital voltage controlled [496](#)
  - voltage-controlled [515](#)
- output variables [79](#)
- overriding parameter values
  - by name [222](#)
  - from the instance statement [222](#)
  - in instances [220](#)
- overview
  - modules [34](#)
  - operators [94](#)
  - system simulation [26](#)
- overview of probes and sources [328](#)

## P

- parallel register model, 8-bit [390](#)
- parallel register, 8-bit [390](#)
- parameters [39](#)
  - aliases [64](#)
  - array values as [225](#)
  - arrays of [62](#)
  - attributes for [59](#)
  - changing during compilation [59](#)
  - changing of cellview bound with an instance [269](#)
  - changing of cellview not currently bound with an instance [270](#)
  - changing value of when bound with an instance [269](#)
  - changing value of when not bound with an instance [270](#)
  - must be constants [59](#)
  - declaration, definition [578](#)
  - declaring [58](#)
  - default value required [61](#)
  - defaults, overriding with Edit Object Properties form [268](#)
  - definition [578](#)
  - deleting from cellviews [271](#)

- dependence on other parameters [59](#)
- enumerated values as [226](#)
- examining current values of [277](#)
- inherited [60](#)
- names [39](#)
- not displayed in Edit Object Properties form unless overridden [278](#)
- overrides
  - detecting [135](#)
- overriding values with module instance statement [222](#)
- permissible values for, specifying [61](#)
- specified in modules, modifying [268](#)
- string [63](#)
- string values as [226](#)
- type specifier optional [61](#)
- type, specifying [61](#)
- parentheses
  - changing evaluation order with [94](#)
  - using to exclude end point in range [62](#)
- parsing, errors during [258](#)
- paths
  - absolute [521](#)
  - relative [521](#)
  - specifying with CDS\_VLOGA\_INCLUDE environment variable [522](#)
- PCM demodulator model, 8-bit [502](#)
- PCM demodulator, 8-bit [502](#)
- PCM modulator model, 8-bit [503](#)
- PCM modulator, 8-bit [503](#)
- period of signal, example of calculating [130](#)
- permissible values for parameters, specifying [61](#)
- permissible values, specifying [61](#)
- phase detector
  - model [504](#)
- phase locked loop model [505](#)
  - digital [495](#)
- pin direction [255](#)
- pins
  - adding to blocks [256](#)
  - deleting [256](#)
  - direction of, in symbols [255](#)
  - specifying information for [262](#)
  - specifying name seed for [255](#)
- PLL model [505](#)
  - digital [495](#)
- PLL, definition [497](#)
- plotting variables [291](#)
- PM demodulator [506](#)

## Cadence Verilog-A Language Reference

---

PM demodulator model [506](#)  
PM modulator [507](#)  
PM modulator model [507](#)  
Poisson distribution [152](#)  
Poisson distribution function [152](#)  
polynomial [431](#)  
polynomial model [431](#)  
port branches [328](#)  
    monitoring flow with [329](#)  
port bus, defining [76](#)  
port connection rules [220](#)  
port declaration example [38](#)  
port direction [37](#)  
port type [37](#)  
ports [36](#)  
    bidirectional [38](#)  
    declaring [36](#)  
    defining by listing nodes [76](#)  
    direction, declaring [37](#)  
    instance, mapping to defining module  
        ports [215](#)  
    names, using to connect instances [220](#)  
    type of, declaring [37](#)  
    undeclared types as [37](#)  
potential  
    definition [579](#)  
    in electrical systems [29](#)  
    probes [328](#)  
    sources, definition [329](#)  
    sources, equivalent circuit model  
        for [330](#)  
    sources, switching to flow sources [331](#)  
potential law. *See* Kirchhoff's Laws [29](#)  
power (pow) function [112](#)  
power consumption, specifying [188](#)  
power electronics components [475](#)  
power function model [432](#)  
power meter model [450](#)  
power sink model, constant [345](#)  
precedence of operators [94](#), [101](#)  
preparing a library [252](#)  
primitives  
    definition [579](#)  
    instantiating in Verilog-A modules [225](#)  
probe model  
    delta [444](#)  
    find event [445](#)  
    signal statistics [445](#), [447](#), [455](#)  
probes [328](#)  
    definition [328](#), [579](#)  
    flow [328](#)

    potential [328](#)  
    reasons for using [328](#)  
procedural assignment statement [82](#)  
procedural assignment statements in the  
    analog block [82](#)  
procedural control constructs [81](#)  
proportional controller model [368](#)  
proportional derivative controller [369](#)  
proportional derivative controller  
    model [369](#)  
proportional integral controller model [370](#)  
proportional integral derivative controller  
    model [371](#)  
protecting  
    all modules in a source description [565](#)  
protection pragmas, using [562](#)  
pump model, charge [491](#)

## Q

Q (charge) meter model [452](#)  
QAM 16-ary demodulator model [508](#)  
QPSK demodulator model [511](#)  
QPSK modulator model [508](#), [512](#)  
QPSK, definition [508](#)  
quadrature amplitude 16-ary modulator  
    model [510](#)  
quadrature phase shift key demodulator  
    model [511](#)  
quadrature phase shift key modulator  
    model [512](#)  
quantities  
    defining [264](#)  
    parameters for [265](#)  
quantity statement  
    modifying absolute tolerances with [529](#)  
    syntax [264](#)  
quantity.spectre file  
    overriding values in [265](#)  
    specifying quantities with [264](#)  
quantizer model [410](#)  
querying the simulation environment [131](#)

## R

random bit stream generator model [513](#)  
random numbers, generating [146](#)  
\$arandom simulator function [146](#)  
\$random simulator function [146](#)

## Cadence Verilog-A Language Reference

---

range  
  for integer numbers [56](#)  
  for real numbers [57](#)  
rate of change, controlling with slew  
  filter [171](#)  
reading from a file [194](#)  
real argument not supported as direction  
  argument [571](#)  
real constants  
  scale factors for [53](#)  
  syntax [52](#)  
real numbers [52](#), [57](#)  
  attributes for [57](#)  
  converting to integers [57](#)  
  declaring [57](#)  
  range permitted [57](#)  
reciprocal model [433](#)  
rectangular hysteresis model [404](#)  
rectifiers  
  behavioral description for [298](#)  
  example [296](#)  
reference directions [29](#)  
  associated [29](#)  
  definition [579](#)  
  illustrated [29](#)  
reference directions, choosing [316](#)  
reference nodes [29](#)  
  compatibility of [220](#)  
  definition [579](#)  
  potential of [29](#)  
related documents [21](#)  
relative paths [521](#)  
relative tolerance [326](#)  
relay  
  example [128](#)  
  model, electromagnetic [393](#)  
reltol (relative tolerance) [326](#)  
repeat loop statement [88](#)  
repeat statement [88](#)  
repeater [411](#)  
repeater model [411](#)  
`resetall compiler directive [243](#)  
resetting directives to default values [243](#)  
resistor model [356](#)  
  self-tuning [349](#)  
  untrimmed [353](#)  
restrainer model [462](#)  
restrictions on using analog operators [161](#)  
rigid branches, attribute for [333](#)  
rise times, setting default for [242](#)  
RLC Circuit [335](#)

RLC circuit [44](#), [45](#)  
RLC circuit model [335](#)  
rms, definition [441](#)  
road model [310](#), [463](#)  
RS-Type Flip-Flop [384](#)  
RS-type flip-flop model [384](#)  
rules, for connecting instances [220](#)  
run time binding, definition [579](#)

## S

s or S format character [186](#)  
sample-and-hold amplifier model  
  (ideal) [472](#)  
sampler model [453](#)  
saturating integrator model [412](#)  
saveahdlvars option [291](#)  
saving Verilog-A variables [291](#)  
scalar node [75](#)  
scale factors, for real constants [53](#)  
schematic cellviews  
  instantiating in Verilog-A  
    components [289](#)  
  opening [269](#)  
  opening in Cadence analog design  
    environment [275](#)  
  rules for instantiating in Verilog-A  
    modules [289](#)  
schHiCreateBlockInst SKILL function [256](#)  
Schottky Diode [485](#)  
Schottky diode model [485](#)  
scope  
  definition [579](#)  
  named block defines new [86](#)  
  of discipline identifiers [71](#)  
  rules [49](#)  
self-tuning resistor [349](#)  
self-tuning resistor model [349](#)  
semiconductor components [478](#)  
sequential block statement [86](#)  
serial register model, 8-bit [391](#)  
serial register, 8-bit [391](#)  
shared object files [531](#)  
shdl\_strchr string function [107](#)  
shdl\_strcspn string function [108](#)  
shdl\_strchr string function [108](#)  
shdl\_strspn string function [109](#)  
shdl\_strstr string function [109](#)  
shifter model, level [383](#), [406](#)  
shock absorber model [309](#)

## Cadence Verilog-A Language Reference

---

- short circuit fault [346](#)
- short circuit fault model [346](#)
- short-circuiting expressions [101](#)
- Show Instance Table button [282](#)
- sigma-delta converter (first-order) [471](#)
- sigma-delta converter model (first order) [471](#)
- signal driver model, differential [401](#)
- signal statistics probe [455](#)
- signal statistics probe model [445](#), [447](#), [455](#)
- signal values
  - modifying with branch contribution statement [83](#)
  - obtaining and setting [138](#)
- signal values, obtaining and setting [137](#)
- signal-flow discipline [72](#)
- signal-flow systems [29](#)
  - modeling supported by Verilog-A [29](#)
  - signal-flow disciplines used to define [77](#)
- signed number [434](#)
- signed number model [434](#)
- signs, requesting in output [185](#)
- simple filename [522](#)
- simple implicit diode [335](#)
- simple implicit diode model [335](#)
- simulating a system [325](#)
- simulation
  - overview [26](#)
  - preparing for [519](#)
- Simulation Environment Options form [289](#)
- simulation environment, querying [131](#)
- simulation time, obtaining current [131](#)
- simulation view lists [289](#)
- Simulator Control Functions [198](#)
- simulator flow [30](#)
- simulator functions
  - \$arandom [146](#)
  - \$dist\_chi\_square [153](#)
  - \$dist\_erlang [154](#)
  - \$dist\_exponential [151](#)
  - \$dist\_normal [150](#)
  - \$dist\_poisson [152](#)
  - \$dist\_t [153](#)
  - \$dist\_uniform [149](#)
  - \$random [146](#)
  - ac\_stim [143](#)
  - analysis [141](#)
  - bound\_step [129](#)
  - discontinuity [127](#)
  - flicker\_noise [144](#)
  - last\_crossing [130](#)
  - noise\_table [145](#)
  - white\_noise [144](#)
- sin function [113](#)
- sine function [113](#)
- single shot model [473](#)
- sinh function [113](#)
- sink model, constant power [345](#)
- sinusoidal source
  - swept, model [413](#)
  - variable frequency, model [416](#)
- sinusoidal stimulus, implementing with ac\_stim [143](#)
- sinusoidal waveforms, controlling with slew filter [172](#)
- sizes, of connected terminals and nodes [220](#)
- SKILL function [256](#)
- SKILL functions,
  - schHiCreateBlockInst [256](#)
- slew filter [171](#)
- slew rate measurement model [449](#), [454](#)
- small-signal AC sources [143](#)
- small-signal noise sources [144](#)
- smoothing piecewise constant waveforms [168](#)
- soft current clamp model [347](#)
- soft voltage clamp model [348](#)
- source model
  - audio [489](#)
  - noise [501](#)
  - swept sinusoidal [413](#)
  - three-phase [414](#)
  - variable frequency sinusoidal [416](#)
- source protection [559](#)
  - ncprotect [559](#)
- sources [329](#)
  - controlled [330](#)
  - current-controlled current [331](#)
  - current-controlled voltage [330](#)
  - definition [328](#), [580](#)
  - flow [329](#)
  - linear conductor model [334](#)
  - linear resistor model [335](#)
  - potential [329](#)
  - reasons for using [328](#)
  - RLC circuit model [335](#)
  - simple implicit diode model [335](#)
  - unassigned [331](#)
  - voltage-controlled current [330](#)
  - voltage-controlled voltage [330](#)
- space

## Cadence Verilog-A Language Reference

---

- displaying or printing [185](#)
  - white [48](#)
- special characters [184](#)
- special characters, displaying [184](#)
- Spectre
  - netlist file [523](#)
  - netlist file, creating [523](#)
  - primitives, instantiating in Verilog-A modules [225](#)
- Spectre/Spectreverilog Interface (Spectre Direct) [265](#)
- SpectreVerilog [251](#)
- SPICE-mode netlisting, naming requirements for [526](#)
- spring model [308](#), [464](#)
- sqr function [112](#)
- square model [435](#)
- square root function [112](#)
- square root model [436](#)
- functions
  - \$sscanf [102](#)
- \$sscanf function [102](#)
- \$sscanf operator
  - details [104](#)
- standard mathematical functions [112](#)
- String [104](#)
- string copy operator [103](#)
- string copy operator, details [103](#)
- string functions
  - \$sscanf [104](#)
  - atoi [106](#)
  - atoreal [106](#)
  - concatenation [103](#)
  - copy [103](#)
  - getc [107](#)
  - len [107](#)
  - shdl\_strchr [107](#)
  - shdl\_strcspn [108](#)
  - shdl\_strrchr [108](#)
  - shdl\_strspn [109](#)
  - shdl\_strstr [109](#)
  - Verilog-A [102](#)
- string parameters, declaring [63](#)
- strings [58](#)
  - atoi operator [102](#)
  - atoreal function [102](#)
  - comparison operators [102](#)
  - comparison operators for [103](#)
  - concatenation operator [102](#)
  - converting to integer [102](#)
  - converting to real [102](#)
  - copy operators [102](#)
  - getc function [102](#)
  - len function [102](#)
  - number of characters in [102](#)
  - operators and functions [101](#)
  - substr function [102](#)
  - substrings off [102](#)
- strings, as parameter values [226](#)
- \$strobe
  - description [184](#), [188](#)
  - example [186](#)
- structural definitions, definition [580](#)
- structural descriptions, undeclared port types in [37](#)
- Student's T distribution [153](#)
- Student's T distribution function [153](#)
- substr function [102](#)
- subtractor model [437](#)
  - four numbers [438](#)
  - full [389](#)
  - half [388](#)
- svcv primitive [225](#)
- swept sinusoidal source
  - model [413](#)
- switch [363](#)
  - branch, creating [84](#)
  - branches [85](#), [331](#), [332](#)
  - branches, value retention for [332](#)
  - model [363](#)
- switch view list
  - illustrated [289](#)
  - modifying with Hierarchy Editor [290](#)
- switched capacitor integrator model [474](#)
- switching the cellview bound with an instance [272](#)
- symbol cellview
  - creating from a new Verilog-A cellview [262](#)
  - creating from a Verilog-A cellview [263](#)
- symbol cellviews, creating from Verilog-A cellviews [263](#)
- symbol editor [256](#)
- Symbol Generation form [262](#)
- symbol view, creating [254](#)
- symbols
  - copying [254](#)
  - creating [254](#), [262](#)
  - creating Verilog-A cellviews from [256](#)
- synchronizing
  - hierarchy editor with changes in the schematic [274](#)

- schematic with changes in the hierarchy
  - editor [273](#)
- syntax
  - braces [23](#)
  - checking, in Cadence analog design environment [258](#)
  - continuation [23](#)
  - definition operator (::=) [22](#)
  - discipline names [23](#)
  - error [517](#)
  - file names [23](#)
  - keywords [23](#)
  - nature names [23](#)
  - square brackets [23](#)
  - variables [23](#)
  - vertical bars [23](#)
- systems
  - conservative [29](#)
  - definition [27](#)

## T

- tab characters
  - as white space [48](#)
  - displaying [184](#)
- table model file format [158](#)
- tan function [113](#)
- tangent function [113](#)
- tanh function [113](#)
- technology file [254](#)
- technology file for New Library form [254](#)
- telecommunications components [486](#)
- temperature, obtaining current
  - ambient [132](#)
- terminals
  - as defined for gearbox [316](#)
  - branch [78](#)
- ternary operator [100](#)
- text editor, using to create modules [261](#)
- text macros [238](#)
  - defining [238](#)
  - restrictions on [239](#)
  - undefining [240](#)
- thermal voltage, obtaining [132](#)
- thin-film MOSFET [301](#)
- thin-film transistor (TFT) model [301](#)
- third-order polynomial function model [431](#)
- three-phase
  - motor model [394](#)
  - source model [414](#)

- thyristor model [477](#)
- thyristors
  - behavioral description for [297](#)
  - compared to diodes [297](#)
- time derivative operator [162](#)
- time integral operator [162](#)
- time step, bounding [129](#)
- time-points, placed by transition filter [168](#)
- timer event [124](#)
- timer function [124](#)
- `timescale compiler directive
  - not reset by `resetall directive [243](#)
  - syntax [242](#)
- toggle-type flip-flop model [385](#)
- tolerances
  - absolute [326](#)
  - relative [326](#)
- top-down design [255](#)
- torque [316](#)
- transformer model, two-phase [421](#)
- transformer, behavioral description for [298](#)
- transient analysis [325](#)
- transistor model
  - MOS (level 1) [479](#)
  - MOS thin-film [481](#)
  - N JFET [482](#)
  - NPN bipolar junction [483](#)
- transition filter
  - not recommended for smoothly varying waveforms [169](#)
  - syntax [168](#)
- transmission channel model [514](#)
- triangular wave source, example [127](#)
- trigger-type flip-flop model [385](#)
- trigonometric and hyperbolic functions [113](#)
- trigonometric functions [113](#)
- troubleshooting loops of rigid
  - branches [333](#)
- two-phase transformer model [421](#)
- type specifier, optional on parameter
  - declaration [61](#)
- typographic and syntax conventions [22](#)

## U

- unary operators [95](#)
  - defined [95](#)
  - precedence of [95](#)
- unary reduction operators [95](#)
- unassigned sources [331](#)

[`undef compiler directive 240](#)  
[undefining text macros 240](#)  
[underscore, in identifiers 49](#)  
[uniform distribution 149](#)  
[uniform distribution function 149](#)  
 unit attribute  
     [description 69](#)  
     [for integers 57, 333](#)  
     [for parameters 59](#)  
     [for reals 57](#)  
     [requirements for 70](#)  
[units \(scale factors\) for real numbers 53](#)  
 untrimmed  
     [capacitor model 351](#)  
     [inductor model 352](#)  
     [resistor model 353](#)  
[user-defined functions 202](#)  
     [calling 207](#)  
     [declaring 202](#)

## V

[.va file extension 520](#)  
[value retention for switch branches 332](#)  
[value-to-flow converter model 415](#)  
[variable frequency sinusoidal source model 416](#)  
[variable-gain amplifier model, voltage-controlled 355](#)  
[variable-gain differential amplifier model 417](#)  
 variables  
     [displaying waveforms of 291](#)  
[variables, choosing 316](#)  
[VCO model 515](#)  
[VCO, definition 495](#)  
[vector nodes, definition 75](#)  
[vectors, arguments represented as 173](#)  
[Verilog and VHDL 290](#)  
 Verilog, digital  
     [cannot instantiate below Verilog-A module 290](#)  
     [wiring to Verilog-A components 290](#)  
 Verilog-A  
     [definition 580](#)  
     [language overview 26](#)  
     [.va extension for files 520](#)  
 Verilog-A cellviews  
     [creating 260](#)  
     [creating from a symbol or block 256](#)

[creating from existing Verilog-A cellviews 267](#)  
     [creating from scratch 260](#)  
     [creating from symbols or blocks 256](#)  
     [editing outside of the analog design environment 258](#)  
[veriloga cellviews, creating with VerilogA-Editor 261](#)  
[Verilog-A instrumentation module 539](#)  
[Verilog-A module description, creating 520](#)  
[Verilog-A string functions, table of 102](#)  
 VHDL  
     [cannot instantiate below Verilog-A module. 290](#)  
     [wiring to Verilog-A components 290](#)  
[voltage clamp model](#)  
     [hard 342](#)  
     [soft 348](#)  
[voltage deadband amplifier 41](#)  
     [model 354](#)  
[voltage meter model 457](#)  
[voltage source model](#)  
     [current-controlled 360](#)  
     [voltage-controlled 359](#)  
[voltage-controlled current source 330](#)  
[voltage-controlled current source model 361](#)  
[voltage-controlled oscillator model 515](#)  
     [model, digital 496](#)  
[voltage-controlled variable-gain amplifier model 355](#)  
[voltage-controlled voltage source 330](#)  
[voltage-controlled voltage source model 359](#)

## W

[waveforms, displaying 291](#)  
[wheel 465](#)  
[wheel model 312, 465](#)  
[while loop statement 89](#)  
[while statement 89](#)  
[white space 48](#)  
[white\\_noise function 144](#)  
[white\\_noise simulator function 144](#)  
[winding model, magnetic 420](#)  
[wire \(predefined empty discipline\) 72](#)  
[writing to a file 195](#)



### X

XNOR Gate [378](#)  
XNOR gate model [378](#)  
XOR Gate [377](#)  
XOR gate model [377](#)

### Z

Z (impedance) meter [458](#)  
Z (impedance) meter model [458](#)  
zero crosses, detecting [120](#)  
zero-denominator Laplace transforms [175](#)  
zero-denominator Z-transforms [180](#)  
zero-pole Laplace transforms [174](#)  
zero-pole Z-transforms [179](#)  
zi\_nd Z-transform filter [182](#)  
zi\_np Z-transform filter [181](#)  
zi\_zd Z-transform filter [180](#)  
zi\_zp Z-transform filter [179](#)  
Z-transform filters [178](#)  
Z-transforms  
    introduction [178](#)  
    numerator-denominator form [182](#)  
    numerator-pole form [181](#)  
    zero-denominator form [180](#)  
    zero-pole form [179](#)

# Cadence Verilog-A Language Reference

---