Contents

Preface ........................................................................................................................................... 19
Related Documents ................................................................................................................... 19
Internet Mail Address ............................................................................................................... 19
Typographic and Syntax Conventions ....................................................................................... 20

1
Modeling Concepts .................................................................................................................... 23
Verilog-A Language Overview .................................................................................................. 24
Describing a System .................................................................................................................. 25
Analog Systems .......................................................................................................................... 26
Nodes ........................................................................................................................................ 26
Conservative Systems .............................................................................................................. 27
Signal-Flow Systems ................................................................................................................. 27
Mixed Conservative and Signal-Flow Systems ........................................................................... 27
Simulator Flow .......................................................................................................................... 28

2
Creating Modules ....................................................................................................................... 31
Overview .................................................................................................................................. 32
Declaring Modules ..................................................................................................................... 32
Declaring the Module Interface ................................................................................................. 33
Module Name ............................................................................................................................ 34
Ports ........................................................................................................................................ 34
Parameters ............................................................................................................................... 36
Defining Module Analog Behavior ............................................................................................. 37
Defining Analog Behavior with Control Flow .............................................................................. 38
Using Integration and Differentiation with Analog Signals ......................................................... 40
Using Internal Nodes in Modules ............................................................................................. 41
Using Internal Nodes in Behavioral Definitions ......................................................................... 41
Using Internal Nodes in Higher Order Systems ........................................................................ 42
3
Lexical Conventions .................................................. 45
White Space .......................................................... 46
Comments ......................................................... 46
Identifiers .......................................................... 46
   Ordinary Identifiers ........................................ 47
   Escaped Names ............................................. 47
   Scope Rules .................................................. 47
Numbers .......................................................... 48
   Integer Numbers ........................................... 48
   Real Numbers ............................................... 48

4
Data Types and Objects ........................................... 51
Integer Numbers .................................................. 52
Real Numbers ..................................................... 52
   Converting Real Numbers to Integer Numbers ........ 53
Strings ............................................................. 54
Parameters .......................................................... 54
   Specifying a Parameter Type ............................... 56
   Specifying Permissible Values .............................. 56
   Specifying Parameter Arrays ............................... 58
Local Parameters .................................................... 59
String Parameters .................................................. 59
Parameter Aliases ................................................... 59
Paramsets .............................................................. 60
   Paramset Output Variables ................................. 61
Genvars ............................................................... 62
Natures ................................................................. 63
   Declaring a Base Nature ..................................... 64
Disciplines ............................................................. 66
   Binding Natures with Potential and Flow ............ 66
   Compatibility of Disciplines ............................. 67
## 5 Statements for the Analog Block

### Assignment Statements
- Procedural Assignment Statements in the Analog Block
  - Branch Contribution Statement
  - Indirect Branch Assignment Statement
- Sequential Block Statement
- Conditional Statement
- Case Statement
- Repeat Statement
- While Statement
- For Statement
- Generate Statement

## 6 Operators for Analog Blocks

### Overview of Operators
- Unary Operators
  - Unary Reduction Operators
- Binary Operators
  - Bitwise Operators
- Ternary Operator
- Operator Precedence
- Expression Short-Circuiting
- String Operators and Functions
  - Mapping SpectreHDL String Functions to Verilog-A Functions
  - String Operator Details
  - String Function Details
# 7 Built-In Mathematical Functions

- Standard Mathematical Functions
- Trigonometric and Hyperbolic Functions
- Controlling How Math Domain Errors Are Handled

# 8 Detecting and Using Analog Events

- Detecting and Using Events
  - Initial step Event
  - Final step Event
  - Cross Event
  - Above Event
  - Timer Event

# 9 Simulator Functions

- Announcing Discontinuity
- Bounding the Time Step
- Announcing and Handling Nonlinearities
- Finding When a Signal Is Zero
- Querying the Simulation Environment
  - Obtaining the Current Simulation Time
  - Obtaining the Current Ambient Temperature
  - Obtaining the Thermal Voltage
  - Querying the scale, gmin, and iteration Simulation Parameters
- Detecting Parameter Overrides
- Obtaining and Setting Signal Values
- Accessing Attributes
- Analysis-Dependent Functions
  - Determining the Current Analysis Type
  - Implementing Small-Signal AC Sources
  - Implementing Small-Signal Noise Sources
- Generating Random Numbers
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generating Random Numbers in Specified Distributions</td>
<td>133</td>
</tr>
<tr>
<td>Uniform Distribution</td>
<td>133</td>
</tr>
<tr>
<td>Normal (Gaussian) Distribution</td>
<td>134</td>
</tr>
<tr>
<td>Exponential Distribution</td>
<td>135</td>
</tr>
<tr>
<td>Poisson Distribution</td>
<td>136</td>
</tr>
<tr>
<td>Chi-Square Distribution</td>
<td>136</td>
</tr>
<tr>
<td>Student's T Distribution</td>
<td>137</td>
</tr>
<tr>
<td>Erlang Distribution</td>
<td>138</td>
</tr>
<tr>
<td>Interpolating with Table Models</td>
<td>139</td>
</tr>
<tr>
<td>Table Model File Format</td>
<td>140</td>
</tr>
<tr>
<td>Example: Using the $table_model Function</td>
<td>142</td>
</tr>
<tr>
<td>Example: Preparing Data in One-Dimensional Array Format</td>
<td>142</td>
</tr>
<tr>
<td>Analog Operators</td>
<td>143</td>
</tr>
<tr>
<td>Restrictions on Using Analog Operators</td>
<td>144</td>
</tr>
<tr>
<td>Limited Exponential Function</td>
<td>144</td>
</tr>
<tr>
<td>Time Derivative Operator</td>
<td>144</td>
</tr>
<tr>
<td>Time Integral Operator</td>
<td>145</td>
</tr>
<tr>
<td>Circular Integrator Operator</td>
<td>147</td>
</tr>
<tr>
<td>Derivative Operator</td>
<td>149</td>
</tr>
<tr>
<td>Delay Operator</td>
<td>150</td>
</tr>
<tr>
<td>Transition Filter</td>
<td>151</td>
</tr>
<tr>
<td>Slew Filter</td>
<td>154</td>
</tr>
<tr>
<td>Implementing Laplace Transform S-Domain Filters</td>
<td>156</td>
</tr>
<tr>
<td>Implementing Z-Transform Filters</td>
<td>161</td>
</tr>
<tr>
<td>Displaying Results</td>
<td>165</td>
</tr>
<tr>
<td>$strobe</td>
<td>166</td>
</tr>
<tr>
<td>$display</td>
<td>169</td>
</tr>
<tr>
<td>$write</td>
<td>169</td>
</tr>
<tr>
<td>$debug</td>
<td>169</td>
</tr>
<tr>
<td>Specifying Power Consumption</td>
<td>170</td>
</tr>
<tr>
<td>Working with Files</td>
<td>171</td>
</tr>
<tr>
<td>Opening a File</td>
<td>171</td>
</tr>
<tr>
<td>Reading from a File</td>
<td>174</td>
</tr>
<tr>
<td>Writing to a File</td>
<td>174</td>
</tr>
<tr>
<td>Closing a File</td>
<td>176</td>
</tr>
<tr>
<td>Exiting to the Operating System</td>
<td>176</td>
</tr>
</tbody>
</table>
10
Instantiating Modules and Primitives .................................................. 181
Instantiating Verilog-A Modules .......................................................... 182
  Creating and Naming Instances ......................................................... 182
  Mapping Instance Ports to Module Ports ............................................ 183
Connecting the Ports of Module Instances .......................................... 184
  Port Connection Rules ...................................................................... 185
Overriding Parameter Values in Instances ............................................ 185
  Overriding Parameter Values from the Instantiation Statement ............. 185
  Overriding Parameter Values by Using Paramsets ................................ 186
Instantiating Analog Primitives ............................................................ 188
  Instantiating Analog Primitives that Use Array Valued Parameters ....... 188
  Instantiating Modules that Use Unsupported Parameter Types ............... 189
Using Inherited Ports ........................................................................... 189
Using an m-factor (Multiplicity Factor) ................................................. 190
  Accessing an Inherited m-factor ......................................................... 191
  Example: Using an m-factor ............................................................. 191

11
Controlling the Compiler ..................................................................... 193
Using Compiler Directives .................................................................... 194
Implementing Text Macros .................................................................... 194
  `define Compiler Directive ................................................................ 194
  `undef Compiler Directive ................................................................ 196
Compiling Code Conditionally ............................................................... 196
Including Files at Compilation Time ..................................................... 196
Setting Default Rise and Fall Times ..................................................... 197
Resetting Directives to Default Values ............................................... 197
Checking the Simulator Version ............................................................ 198
Checking Support for Compact Modeling Extensions ............................ 198
12
Using Verilog-A in the Cadence Analog Design Environment

201

Creating Cellviews Using the Cadence Analog Design Environment .................. 202
  Preparing a Library ...................................................................................... 202
  Creating the Symbol View ................................................................. 204
  Using Blocks ............................................................................................ 205
  Creating a Verilog-A Cellview from a Symbol or Block ......................... 206
  Descend Edit ............................................................................................ 209
  Creating a Verilog-A Cellview ............................................................ 210
  Creating a Symbol Cellview from an Analog HDL Cellview ................. 212
Using Escaped Names in the Cadence Analog Design Environment .......... 214
Defining Quantities ..................................................................................... 214
  spectre/spectreVerilog Interface (Spectre Direct) .................................... 215
Using Multiple Cellviews for Instances ...................................................... 216
  Creating Multiple Cellviews for a Component .......................................... 217
  Modifying the Parameters Specified in Modules .................................... 218
  Switching the Cellview Bound with an Instance ..................................... 221
  Example Illustrating Cellview Switching ............................................. 225
Multilevel Hierarchical Designs ................................................................. 234
  Including Verilog-A through Model Setup ........................................... 235
  Netlisting Verilog-A Modules ............................................................. 235
  Hierarchical Verilog-A Modules .......................................................... 235
  Using a Hierarchy ..................................................................................... 237
Using Models with Verilog-A .................................................................... 239
  Models in Modules .................................................................................. 239
Saving Verilog-A Variables ....................................................................... 240
Displaying the Waveforms of Variables .................................................... 240

13
Advanced Modeling Examples ................................................................. 243
Electrical Modeling ...................................................................................... 243
  Three-Phase, Half-Wave Rectifier ....................................................... 243
  Thin-Film Transistor Model ............................................................... 249
Mechanical Modeling .......................................................... 255
  Car on a Bumpy Road .................................................. 256
  Gearbox .................................................................... 263

A
Nodal Analysis ............................................................... 269
  Kirchhoff’s Laws .......................................................... 270
  Simulating a System .................................................... 271
    Transient Analysis ................................................... 271
    Convergence ................................................................ 271

B
Analog Probes and Sources .............................................. 273
  Overview of Probes and Sources .................................. 274
  Probes ........................................................................ 274
  Port Branches ............................................................... 275
  Sources ....................................................................... 275
    Unassigned Sources .................................................. 277
    Switch Branches ......................................................... 277
  Examples of Sources and Probes .................................. 280
    Linear Conductor ....................................................... 280
    Linear Resistor .......................................................... 281
    RLC Circuit ................................................................ 281
    Simple Implicit Diode .................................................. 281

C
Standard Definitions ....................................................... 283
  disciplines.vams File .................................................... 284
  constants.vams File ....................................................... 288

D
Sample Model Library ...................................................... 291
  Analog Components ..................................................... 293
Analog Multiplexer ............................................................. 293
Current Deadband Amplifier ........................................... 294
Hard Current Clamp .......................................................... 295
Hard Voltage Clamp ............................................................ 296
Open Circuit Fault ............................................................... 297
Operational Amplifier .......................................................... 298
Constant Power Sink ........................................................... 299
Short Circuit Fault ............................................................... 300
Soft Current Clamp .............................................................. 301
Soft Voltage Clamp .............................................................. 302
Self-Tuning Resistor ............................................................. 303
Untrimmed Capacitor ........................................................... 305
Untrimmed Inductor ............................................................ 306
Untrimmed Resistor ............................................................ 307
Voltage Deadband Amplifier ............................................... 308
Voltage-Controlled Variable-Gain Amplifier ....................... 309
Basic Components ............................................................... 310
Resistor ........................................................................ 310
Capacitor ........................................................................ 311
Inductor .......................................................................... 312
Voltage-Controlled Voltage Source .................................... 313
Current-Controlled Voltage Source .................................... 314
Voltage-Controlled Current Source .................................... 315
Current-Controlled Current Source .................................... 316
Switch ............................................................................ 317
Control Components ............................................................ 318
Error Calculation Block ....................................................... 318
Lag Compensator ................................................................. 319
Lead Compensator ............................................................... 320
Lead-Lag Compensator ......................................................... 321
Proportional Controller ....................................................... 322
Proportional Derivative Controller ..................................... 323
Proportional Integral Controller ......................................... 324
Proportional Integral Derivative Controller ......................... 325
Logic Components ............................................................... 326
AND Gate ..................................................................... 326

Cadence Verilog-A Language Reference
NAND Gate ................................................................. 327
OR Gate ................................................................. 328
NOT Gate ................................................................. 329
NOR Gate ................................................................. 330
XOR Gate ................................................................. 331
XNOR Gate ............................................................... 332
D-Type Flip-Flop ...................................................... 333
Clocked JK Flip-Flop .................................................. 334
JK-Type Flip-Flop ...................................................... 336
Level Shifter ............................................................ 337
RS-Type Flip-Flop ...................................................... 338
Trigger-Type (Toggle-Type) Flip-Flop ......................... 339
Half Adder ............................................................... 340
Full Adder ............................................................... 341
Half Subtractor ........................................................ 342
Full Subtractor ......................................................... 343
Parallel Register, 8-Bit .............................................. 344
Serial Register, 8-Bit .................................................. 345
Electromagnetic Components ................................. 346
DC Motor ............................................................... 346
Electromagnetic Relay ............................................... 347
Three-Phase Motor ................................................... 348
Functional Blocks .................................................... 349
Amplifier ............................................................... 349
Comparator ............................................................ 350
Controlled Integrator ............................................... 351
Deadband ............................................................... 352
Deadband Differential Amplifier .............................. 353
Differential Amplifier (Opamp) ................................. 354
Differential Signal Driver ......................................... 355
Differentiator .......................................................... 356
Flow-to-Value Converter .......................................... 357
Rectangular Hysteresis ............................................. 358
Integrator ............................................................... 359
Level Shifter ........................................................... 360
Limiting Differential Amplifier ................................. 361
<table>
<thead>
<tr>
<th>Component Type</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logarithmic Amplifier</td>
<td>362</td>
</tr>
<tr>
<td>Multiplexer</td>
<td>363</td>
</tr>
<tr>
<td>Quantizer</td>
<td>364</td>
</tr>
<tr>
<td>Repeater</td>
<td>365</td>
</tr>
<tr>
<td>Saturating Integrator</td>
<td>366</td>
</tr>
<tr>
<td>Swept Sinusoidal Source</td>
<td>367</td>
</tr>
<tr>
<td>Three-Phase Source</td>
<td>368</td>
</tr>
<tr>
<td>Value-to-Flow Converter</td>
<td>369</td>
</tr>
<tr>
<td>Variable Frequency Sinusoidal Source</td>
<td>370</td>
</tr>
<tr>
<td>Variable-Gain Differential Amplifier</td>
<td>371</td>
</tr>
<tr>
<td>Magnetic Components</td>
<td>372</td>
</tr>
<tr>
<td>Magnetic Core</td>
<td>372</td>
</tr>
<tr>
<td>Magnetic Gap</td>
<td>373</td>
</tr>
<tr>
<td>Magnetic Winding</td>
<td>374</td>
</tr>
<tr>
<td>Two-Phase Transformer</td>
<td>375</td>
</tr>
<tr>
<td>Mathematical Components</td>
<td>376</td>
</tr>
<tr>
<td>Absolute Value</td>
<td>376</td>
</tr>
<tr>
<td>Adder</td>
<td>377</td>
</tr>
<tr>
<td>Adder, 4 Numbers</td>
<td>378</td>
</tr>
<tr>
<td>Cube</td>
<td>379</td>
</tr>
<tr>
<td>Cubic Root</td>
<td>380</td>
</tr>
<tr>
<td>Divider</td>
<td>381</td>
</tr>
<tr>
<td>Exponential Function</td>
<td>382</td>
</tr>
<tr>
<td>Multiplier</td>
<td>383</td>
</tr>
<tr>
<td>Natural Log Function</td>
<td>384</td>
</tr>
<tr>
<td>Polynomial</td>
<td>385</td>
</tr>
<tr>
<td>Power Function</td>
<td>386</td>
</tr>
<tr>
<td>Reciprocal</td>
<td>387</td>
</tr>
<tr>
<td>Signed Number</td>
<td>388</td>
</tr>
<tr>
<td>Square</td>
<td>389</td>
</tr>
<tr>
<td>Square Root</td>
<td>390</td>
</tr>
<tr>
<td>Subtractor</td>
<td>391</td>
</tr>
<tr>
<td>Subtractor, 4 Numbers</td>
<td>392</td>
</tr>
<tr>
<td>Measure Components</td>
<td>393</td>
</tr>
<tr>
<td>ADC, 8-Bit Differential Nonlinearity Measurement</td>
<td>393</td>
</tr>
<tr>
<td>ADC, 8-Bit Integral Nonlinearity Measurement</td>
<td>394</td>
</tr>
<tr>
<td>Component Type</td>
<td>Page</td>
</tr>
<tr>
<td>---------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Ammeter (Current Meter)</td>
<td>395</td>
</tr>
<tr>
<td>DAC, 8-Bit Differential Nonlinearity Measurement</td>
<td>396</td>
</tr>
<tr>
<td>DAC, 8-Bit Integral Nonlinearity Measurement</td>
<td>397</td>
</tr>
<tr>
<td>Delta Probe</td>
<td>398</td>
</tr>
<tr>
<td>Find Event Probe</td>
<td>399</td>
</tr>
<tr>
<td>Find Slope</td>
<td>401</td>
</tr>
<tr>
<td>Frequency Meter</td>
<td>402</td>
</tr>
<tr>
<td>Offset Measurement</td>
<td>403</td>
</tr>
<tr>
<td>Power Meter</td>
<td>404</td>
</tr>
<tr>
<td>Q (Charge) Meter</td>
<td>406</td>
</tr>
<tr>
<td>Sampler</td>
<td>407</td>
</tr>
<tr>
<td>Slew Rate Measurement</td>
<td>408</td>
</tr>
<tr>
<td>Signal Statistics Probe</td>
<td>409</td>
</tr>
<tr>
<td>Voltage Meter</td>
<td>411</td>
</tr>
<tr>
<td>Z (Impedance) Meter</td>
<td>412</td>
</tr>
<tr>
<td>Mechanical Systems</td>
<td>413</td>
</tr>
<tr>
<td>Gearbox</td>
<td>413</td>
</tr>
<tr>
<td>Mechanical Damper</td>
<td>414</td>
</tr>
<tr>
<td>Mechanical Mass</td>
<td>415</td>
</tr>
<tr>
<td>Mechanical Restrainer</td>
<td>416</td>
</tr>
<tr>
<td>Road</td>
<td>417</td>
</tr>
<tr>
<td>Mechanical Spring</td>
<td>418</td>
</tr>
<tr>
<td>Wheel</td>
<td>419</td>
</tr>
<tr>
<td>Mixed-Signal Components</td>
<td>420</td>
</tr>
<tr>
<td>Analog-to-Digital Converter, 8-Bit</td>
<td>420</td>
</tr>
<tr>
<td>Analog-to-Digital Converter, 8-Bit (Ideal)</td>
<td>421</td>
</tr>
<tr>
<td>Decimator</td>
<td>422</td>
</tr>
<tr>
<td>Digital-to-Analog Converter, 8-Bit</td>
<td>423</td>
</tr>
<tr>
<td>Digital-to-Analog Converter, 8-Bit (Ideal)</td>
<td>424</td>
</tr>
<tr>
<td>Sigma-Delta Converter (first-order)</td>
<td>425</td>
</tr>
<tr>
<td>Sample-and-Hold Amplifier (Ideal)</td>
<td>426</td>
</tr>
<tr>
<td>Single Shot</td>
<td>427</td>
</tr>
<tr>
<td>Switched Capacitor Integrator</td>
<td>428</td>
</tr>
<tr>
<td>Power Electronics Components</td>
<td>429</td>
</tr>
<tr>
<td>Full Wave Rectifier, Two Phase</td>
<td>429</td>
</tr>
<tr>
<td>Half Wave Rectifier, Two Phase</td>
<td>430</td>
</tr>
</tbody>
</table>
# Cadence Verilog-A Language Reference

<table>
<thead>
<tr>
<th>Component</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thyristor</td>
<td>431</td>
</tr>
<tr>
<td>Semiconductor Components</td>
<td>432</td>
</tr>
<tr>
<td>Diode</td>
<td>432</td>
</tr>
<tr>
<td>MOS Transistor (Level 1)</td>
<td>433</td>
</tr>
<tr>
<td>MOS Thin-Film Transistor</td>
<td>435</td>
</tr>
<tr>
<td>N JFET Transistor</td>
<td>436</td>
</tr>
<tr>
<td>NPN Bipolar Junction Transistor</td>
<td>437</td>
</tr>
<tr>
<td>Schottky Diode</td>
<td>439</td>
</tr>
<tr>
<td>Telecommunications Components</td>
<td>440</td>
</tr>
<tr>
<td>AM Demodulator</td>
<td>440</td>
</tr>
<tr>
<td>AM Modulator</td>
<td>441</td>
</tr>
<tr>
<td>Attenuator</td>
<td>442</td>
</tr>
<tr>
<td>Audio Source</td>
<td>443</td>
</tr>
<tr>
<td>Bit Error Rate Calculator</td>
<td>444</td>
</tr>
<tr>
<td>Charge Pump</td>
<td>445</td>
</tr>
<tr>
<td>Code Generator, 2-Bit</td>
<td>446</td>
</tr>
<tr>
<td>Code Generator, 4-Bit</td>
<td>447</td>
</tr>
<tr>
<td>Decider</td>
<td>448</td>
</tr>
<tr>
<td>Digital Phase Locked Loop (PLL)</td>
<td>449</td>
</tr>
<tr>
<td>Digital Voltage-Controlled Oscillator</td>
<td>450</td>
</tr>
<tr>
<td>FM Demodulator</td>
<td>451</td>
</tr>
<tr>
<td>FM Modulator</td>
<td>452</td>
</tr>
<tr>
<td>Frequency-Phase Detector</td>
<td>453</td>
</tr>
<tr>
<td>Mixer</td>
<td>454</td>
</tr>
<tr>
<td>Noise Source</td>
<td>455</td>
</tr>
<tr>
<td>PCM Demodulator, 8-Bit</td>
<td>456</td>
</tr>
<tr>
<td>PCM Modulator, 8-Bit</td>
<td>457</td>
</tr>
<tr>
<td>Phase Detector</td>
<td>458</td>
</tr>
<tr>
<td>Phase Locked Loop</td>
<td>459</td>
</tr>
<tr>
<td>PM Demodulator</td>
<td>460</td>
</tr>
<tr>
<td>PM Modulator</td>
<td>461</td>
</tr>
<tr>
<td>QAM 16-ary Demodulator</td>
<td>462</td>
</tr>
<tr>
<td>Quadrature Amplitude 16-ary Modulator</td>
<td>464</td>
</tr>
<tr>
<td>QPSK Demodulator</td>
<td>465</td>
</tr>
<tr>
<td>QPSK Modulator</td>
<td>466</td>
</tr>
<tr>
<td>Random Bit Stream Generator</td>
<td>467</td>
</tr>
</tbody>
</table>
## Cadence Verilog-A Language Reference

**E**

**Verilog-A Keywords** .................................................. 471

**Keywords to Support Backward Compatibility** ....................... 473

**F**

**Understanding Error Messages** ....................................... 475

**G**

**Getting Ready to Simulate** ........................................... 477

Creating a Verilog-A Module Description .......................... 478

  - File Extension .va ........................................... 478
  - include Compiler Directive .................................. 478

Creating a Spectre Netlist File ...................................... 480

  - Including Files in a Netlist ................................ 481
  - Naming Requirements for SPICE-Mode Netlisting ........... 483

Modifying Absolute Tolerances ....................................... 483

  - Modifying abstol in Standalone Mode ....................... 484
  - Modifying abstol in the Cadence Analog Design Environment .... 485

Using the Compiled C Code Flow .................................... 487

  - Turning the Compiled C Code Flow Off and On ............ 487
  - Creating and Specifying Compiled C Code Databases ....... 488
  - Reusing and Sharing Compiled C Objects .................. 488

**H**

**Supported and Unsupported Language Elements** ................. 491

**I**

**Updating Verilog-A Modules** ....................................... 495

  - Suggestions for Updating Models ............................ 496
Current Probes ................................................................. 496
Analog Functions ............................................................. 497
NULL Statements ............................................................... 497
inf Used as a Number ......................................................... 498
Changing Delay to Absdelay ............................................... 498
Changing $realtime to $abstime .......................................... 498
Changing bound_step to $bound_step ................................ 498
Changing Array Specifications ............................................ 499
Chained Assignments Made Illegal ..................................... 499
Real Argument Not Supported as Direction Argument ............ 499
$limexp Changed to limeexp .............................................. 499
`if `MACRO is Not Allowed .............................................. 500
$warning is Not Allowed ................................................... 500
discontinuity Changed to $discontinuity ............................. 500

J
Creating ViewInfo for Verilog-A Cellview .......................... 501
ahdlUpdateViewInfo ......................................................... 501
  Description ............................................................... 501
  Arguments ............................................................... 501
  Example 1 ............................................................... 501
  Example 2 ............................................................... 502
  Example 3 ............................................................... 502

Glossary ................................................................. 503

Index ................................................................. 509
Preface

This manual describes the Cadence® Verilog®-A language, 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. The guidance given here is designed for users who are familiar with the development, design, and simulation of circuits and with high-level programming languages, such as C.

The preface discusses the following:

- Related Documents on page 19
- Internet Mail Address on page 19
- Typographic and Syntax Conventions on page 20

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
- Verilog-A Debugging Tool User Guide
- Cadence Hierarchy Editor User Guide
- Instance-Based View Switching Application Note
- Virtuoso Spectre Circuit Simulator Reference
- Virtuoso Spectre Circuit Simulator User Guide

Internet Mail Address

You can send product enhancement requests and report obscure problems to Customer Support. For current phone numbers and e-mail addresses, see
http://sourcelink.cadence.com/supportcontacts.html

For help with obscure problems, please include the following in your e-mail:

- The license server host ID
  To determine what your server's host ID is, use the SourceLink® Subscription Service http://Sourcelink.cadence.com/hostid/ for assistance.
- A description of the problem
- The version of the product that you are using
- A netlist and all included files including Verilog-A modules so that Customer Support can reproduce the problem
- Output logs and error messages

**Typographic and Syntax Conventions**

Special typographical conventions are used to emphasize or distinguish certain kinds of text in this document. The formal syntax used in this reference uses the definition operator, :=, to define the more complex elements of the Verilog-A language in terms of less complex elements.

- Lowercase words represent syntactic categories. For example,
  ```verbatim
  module_declaration
  ```
  Some names begin with a part that indicates how the name is used. For example,
  ```verbatim
  node_identifier
  ```
  represents an identifier that is used to declare or reference a node.

- Boldface words represent elements of the syntax that must be used exactly as presented. Such items include keywords, operators, and punctuation marks. For example,
  ```verbatim
  endmodule
  ```

- Vertical bars indicate alternatives. You can choose to use any one of the items separated by the bars. For example,
  ```verbatim
  attribute ::= abstol access ddt_nature idt_nature units
  ```
Square brackets enclose optional items. For example,

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

Braces enclose an item that can be repeated zero or more times. For example,

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

Code examples are displayed in Courier font.

```plaintext
/* This is an example of Courier font.*/
```

Within the text, the variables are in Courier italic.

```
This is an example of the Courier italic font.
```

Within the text, the keywords, filenames, names of natures, and names of disciplines are set in Courier 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 is indented on the next line, like this:

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

To distinguish Verilog-A module descriptions from netlists, the netlists are enclosed in boxes and include a comment line at the beginning identifying them as netlists. Here is a sample netlist:

```plaintext
// sample circuit netlist
simulator lang=spectre
global gnd
ahdl_include "description.va"
vin1 in gnd vsource type=sine freq=1e3 ampl=1
    hdlmodule in gnd out opamp gain=2e5
tranAnal tran stop=10e-4
```
Modeling Concepts

This chapter introduces some important concepts basic to using the Cadence® Verilog®-A language, including

- Verilog-A Language Overview on page 24
- Describing a System on page 25
- Analog Systems on page 26
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® 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 A, “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
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**

![System diagram](image)

**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 270.

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 29. For more details about how the simulator uses Kirchhoff’s laws, see “Simulating a System” on page 271.
Simulator Flow

Start analysis
\[ t = 0 \]
\[ v(0) = v_0 \]

Update time
\[ t = t + \Delta t \]

Update values
\[ v = v + \Delta v \]

Evaluate equations
\[ f(v,t) = residue \]

Converged?
\[ residue < \epsilon \]
\[ \Delta v < \Delta \]

Accept the time step?

Done?
\[ T = t \]

End
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 32
- Declaring the Module Interface on page 33
- Defining Module Analog Behavior on page 37
- Using Internal Nodes in Modules on page 41
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`.

```verbatim
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);

real reff;
endmodule
```

Interface declarations

Behavioral description

Declaring Modules

To declare a module, use this syntax.

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

module_keyword ::= 
                   module
                   | macromodule

module_items ::= 
               { module_item }
               | analog_block

module_item ::= 
              module_item_declaration 
              | module_instantiation

module_item_declaration ::= 
                         parameter_declaration 
                         | aliasparam_declaration 
                         | input_declaration 
                         | output_declaration 
                         | inout_declaration
```

December 2006 32 Product Version 6.1
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 34.

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

<table>
<thead>
<tr>
<th>For information about</th>
<th>Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog blocks</td>
<td>“Defining Module Analog Behavior” on page 37</td>
</tr>
<tr>
<td>Parameter overrides</td>
<td>“Overriding Parameter Values in Instances” on page 185</td>
</tr>
<tr>
<td>Module instantiation</td>
<td>“Instantiating Verilog-A Modules” on page 182</td>
</tr>
<tr>
<td>Parameter declarations</td>
<td>“Parameters” on page 54</td>
</tr>
<tr>
<td>Input, output, and inout declarations</td>
<td>“Port Direction” on page 35</td>
</tr>
<tr>
<td>Integer declarations</td>
<td>“Integer Numbers” on page 52</td>
</tr>
<tr>
<td>Net discipline declarations</td>
<td>“Net Disciplines” on page 70</td>
</tr>
<tr>
<td>Real declarations</td>
<td>“Real Numbers” on page 52</td>
</tr>
<tr>
<td>Genvar declarations</td>
<td>“Genvars” on page 62</td>
</tr>
<tr>
<td>Analog function declarations</td>
<td>“User-Defined Functions” on page 177</td>
</tr>
</tbody>
</table>

**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® 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 }

port ::=  
    port_expression

port_expression ::=  
    port_identifier
    port_identifier [ constant_expression ]
    port_identifier [ constant_range ]

constant_range ::=  
    msb_constant_expression : lsb_constant_expression

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 70.

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.

```plaintext
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 ]
```

- **input** declares that the signals on the port cannot be set, although they can be used in expressions.
- **output** declares that the signals on the port can be set, but they cannot be used in expressions.
- **inout** declares that the port is bidirectional. The signals on the port can be both set and used in expressions. **inout** 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`.

You cannot use parameters to define `constant_expression`.

**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.

```verilog
module sdiode(np, nn);
  inout np, nn;
  electrical np, nn;
  parameter real area=1;
  parameter real is=1e-14;
  parameter real n=2;
  parameter real cjo=0;
  parameter real m=0.5;
  parameter real phi=0.7;
  parameter real tt=1p;
  real vd, id, qd;

analog begin
  vd = V(np, nn);
  id = area*is*(exp(vd/(n*$vt)) - 1);
  qd = tt*id + area*vd
    *cjo/pow((1 - vd/phi), m);
  I(np, nn) <+ id + ddt(qd);
end
endmodule
```

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” on page 54.

### 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.

```
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_statement can appear only within the analog block.

analog_seq_block are discussed in “Sequential Block Statement” on page 79.

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:

\[ V(n_1, n_2) <+ \text{expression}; \]
\[ I(n_1, n_2) <+ \text{expression}; \]

where \( V(n_1, n_2) \) and \( I(n_1, n_2) \) represent potential and flow sources, respectively. You can define \( \text{expression} \) 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.

```verilog
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.

```verilog
module setvolts(outvolt);
output outvolt;
voltage outvolt;
    analog
        V(outvolt) <+ 5.0;
endmodule
```

**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
    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.

![Graph showing response of vdba module]

**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.

```verilog
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 \( V(\text{in}) \). Then the module sets \( V(\text{out}) \) equal to the derivative of \( V(\text{diff}) \), in effect taking the second order derivative of \( V(\text{in}) \).

```verilog
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));
  end
endmodule
```
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 144, “Circular Integrator Operator” on page 147, and “Time Integral Operator” on page 145.

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.

```verilog
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

\[
V_{out} = V_{in} - R \cdot C \cdot V_{out} - L \cdot C \cdot \dot{V}_{out}
\]

If you set

\[
V_{n1} = \dot{V}_{out}
\]

you can write

\[
V_{out} = V_{in} - R \cdot C \cdot V_{n1} - L \cdot C \cdot \ddot{V}_{n1}
\]

Module `rlc_high_order` implements these descriptions.

```verilog
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 G, “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 12, “Using Verilog-A in the Cadence Analog Design Environment.”
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 46
- **Comments** on page 46
- **Identifiers** on page 46
- **Numbers** on page 48

See also

- **Operators for Analog Blocks** on page 87
- The information about strings in **Displaying Results** on page 165
- **Verilog-A Keywords** on page 471
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® 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

\[
\text{integer_number ::= [ sign ] unsign_num}
\]

- **sign ::=**
  - \(+\) | \(-\)
- **unsign_num ::=**
  - \(\text{decimal_digit \{ _ | decimal_digit \}}\)
- **decimal_digit ::=**
  - \(0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9\)

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.

Examples of integer constants include

\[
\begin{align*}
277195000 \\
277_195_000 & \quad \text{//Same as the previous number} \\
-634 & \quad \text{//A negative number} \\
0005
\end{align*}
\]

**Real Numbers**

The syntax for a real constant is

\[
\]

- **sign ::=**
  - \(+\) | \(-\)
- **unsign_num ::=**
  - \(\text{decimal_digit \{ _ | decimal_digit \}}\)
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.

<table>
<thead>
<tr>
<th>unit_letter</th>
<th>Scale factor</th>
<th>unit_letter</th>
<th>Scale factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>$10^{12}$</td>
<td>k</td>
<td>$10^{3}$</td>
</tr>
<tr>
<td>G</td>
<td>$10^{9}$</td>
<td>m</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>M</td>
<td>$10^{6}$</td>
<td>u</td>
<td>$10^{-6}$</td>
</tr>
<tr>
<td>K</td>
<td>$10^{3}$</td>
<td>n</td>
<td>$10^{-9}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>p</td>
<td>$10^{-12}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>f</td>
<td>$10^{-15}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
<td>$10^{-18}$</td>
</tr>
</tbody>
</table>

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-12
- 1.2G // 1.2 * 10e9
- 213_116.223_642

For information on converting real numbers to integer numbers, see “Converting Real Numbers to Integer Numbers” on page 53.
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.

- Integer Numbers on page 52
- Real Numbers on page 52
- Parameters on page 54
- Local Parameters on page 59
- String Parameters on page 59
- Parameter Aliases on page 59
- Paramsets on page 60
- Genvars on page 62
- Natures on page 63
- Disciplines on page 66
- Net Disciplines on page 70
- Named Branches on page 72
- Implicit Branches on page 73
- Digital Nets and Registers
Data Types and Objects

Integer Numbers

Use the integer declaration to declare variables of type integer.

```verilog
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.

```verilog
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,

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

Although the desc and units attributes are allowed, Cadence tools, in this release, do nothing with the information.

Real Numbers

Use the real declaration to declare variables of type real.

```verilog
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.

```verilog
class a[1:64]; // Declares array of 64 reals
class b, c, d[-20:0]; // Declares 2 reals and an array of reals
class parameter integer min_size = 1, max_size = 30;
class 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,

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

Although the `desc` and `units` attributes are allowed, Cadence tools, in this release, do nothing with the information.

### 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.

```verilog
integer intvalA, intvalB, intvalC;
class 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.

```verilog
real realvar;
realvar = 9.0;
realvar = 2/3 * realvar; // realvar is 9.0, not 6.0
```

In this example, both 2 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.

```language-verilog
string_declaration ::= string list_of_identifiers ;
list_of_identifiers ::= variable_identifier { , variable_identifier}
var_name ::= variable_identifier
```

A string is defined as follows:

```language-verilog
string ::= " { Any_ASCII_character_except_newline } "
```

For example,

```language-verilog
string tmpString, difString;
tmpString="Temporary string";
difString="Different string";
```

Parameters

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

```language-verilog
parameter_declaration ::= parameter [opt_type] list_of_param_assignments ;
```

```language-verilog
opt_type ::= real |
integer |
string
list_of_param_assignments ::= declarator_init { , declarator_init }
deaclator_init ::= parameter_id = constant_exp { opt_value_range }
```

`opt_type` is described in “Specifying a Parameter Type” on page 56. Note that for parameter arrays, however, you must specify a type.

`opt_value_range` is described in “Specifying Permissible Values” on page 56.

`parameter_id` is the name of a parameter being declared.

`param_array_init` is described in “Specifying Parameter Arrays” on page 58.

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 185 for more information.

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

```verilog
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,

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

Although the desc and units attributes are allowed, Cadence tools, in this release, do nothing with the information.

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 instantiation statement for the module. However, an ordinary parameter whose values is set by referring to an inherited parameter can be changed by the instantiation statement.

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

```verilog
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_rangeSpecifier 
    | exclude value_rangeSpecifier 
    | exclude value_constant_expression

value_rangeSpecifier ::= 
    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 declaration ("Parameters" on page 54) to specify information for parameter arrays.

\[
\text{parameter_array_init ::= } \\
\text{parameter_array_id range = constant_param_arrayinit \{opt_value_range\}} \\
\text{range ::= } \\
\text{[ constant_expression : constant_expression ]} \\
\text{constant_param_arrayinit ::= } \\
\text{\{} \text{param_arrayinit_element_list} \text{\}} \\
\text{\'' \{ param_arrayinit_element_list \} \text{\}} \\
\text{\'' \{ replicator_element_list \}} \\
\text{param_arrayinit_element_list ::= } \\
\text{constant_expression \{} constant_expression \}} \\
\text{replicator_element_list ::= } \\
\text{replicator_constant_expression \{} constant_expression \}} \\
\text{\}} \\
\text{\text{parameter_array_id} is the name of a parameter array being declared.} \\
\text{\text{opt_value_range} is described in "Specifying Permissible Values" on page 56.} \\
\text{\text{replicator_constant_expression} is an integer constant with a value greater than} \\
\text{zero that specifies the number of times the associated constant_expression is to be} \\
\text{included in the element list.} \\
\text{For example, parameter arrays might be declared and used as follows:} \\
\text{parameter integer IVgc_length = 4;} \\
\text{parameter real} \\
\text{I_gc[1:IVgc_length] = \'4\{0.00\};} \\
\text{V_gc[1:IVgc_length] = \'\{-5.00, -1.00, 5.00, 10.00\};} \\
\text{Parameter arrays are subject to the following restrictions:} \\
\text{\text{\textbullet} The type of a parameter array must be specified in the declaration.} \\
\text{\textbullet} An array assigned to an instance of a module must be of the exact size of the array} \\
\text{bounds of that instance.} \\
\text{\textbullet} If the array size is changed via a parameter assignment, the parameter array must be} \\
\text{assigned an array of the new size from the same module as the parameter assignment} \\
\text{that changed the parameter array size.} \]
Local Parameters

Use the localparam declaration to specify the parameters of a module.

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

Local parameters cannot be directly modified by ordered or named parameter value assignments.

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 `tmdatal` and gives it the value `table1.dat`.

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

This parameter can, for example, be used to specify the data file for the `$table_model` function.

```
analog begin
  I(d, s) <= $table_model (V(g, s), V(d, s), tmdatal, "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.

```verilog
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.

```verilog
nmos #(trise()) 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 186.

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

```verilog
paramset_item_declaration ::= {attribute_instance} parameter_declaration |
{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
```

```verilog
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);
inout [0:1] in;
output [0:1] out;
electrical [0:1] in;
electrical [0:1] out;
genvar i, j;
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
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.

Each of your nature declarations must

- Be named with a unique identifier
- Include all the required attributes listed in Table 4-3 on page 65.
- 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.

```verilog
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 65. You can also declare user-defined attributes by declaring them just as you declare the predefined attributes. The Spectre® circuit 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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstol</td>
<td>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 “Convergence” on page 271.</td>
</tr>
<tr>
<td>access</td>
<td>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.</td>
</tr>
<tr>
<td>units</td>
<td>Specifies the units to be used for the value accessed by the access function.</td>
</tr>
<tr>
<td>idt_nature</td>
<td>Specifies a nature to apply when the <code>idt</code> or <code>idtmod</code> operators are used. <strong>Note</strong>: This release of Verilog-A ignores this attribute.</td>
</tr>
<tr>
<td>ddt_nature</td>
<td>Specifies a nature to apply when the <code>ddt</code> operator is used. <strong>Note</strong>: This release of Verilog-A ignores this attribute.</td>
</tr>
</tbody>
</table>
The next table describes the Cadence-specific attributes.

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

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>huge</td>
<td>Specifies the maximum change in signal value allowed during a single iteration. The simulator uses huge to facilitate convergence when signal values are very large. Default: 45.036e06</td>
</tr>
<tr>
<td>blowup</td>
<td>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</td>
</tr>
<tr>
<td>maxdelta</td>
<td>Specifies the maximum change allowed on a Newton-Raphson iteration. Default: 0.3</td>
</tr>
</tbody>
</table>

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

**Table 4-3 Attribute Requirements**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Required or optional?</th>
<th>The constant expression must be</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstol</td>
<td>Required</td>
<td>A real value</td>
</tr>
<tr>
<td>access</td>
<td>Required for all base natures</td>
<td>An identifier</td>
</tr>
<tr>
<td>units</td>
<td>Required for all base natures</td>
<td>A string</td>
</tr>
<tr>
<td>idt_nature</td>
<td>Optional</td>
<td>The name of a nature defined elsewhere</td>
</tr>
<tr>
<td>ddt_nature</td>
<td>Optional</td>
<td>The name of a nature defined elsewhere</td>
</tr>
<tr>
<td>huge</td>
<td>Optional</td>
<td>A real value</td>
</tr>
<tr>
<td>blowup</td>
<td>Optional</td>
<td>A real value</td>
</tr>
<tr>
<td>maxdelta</td>
<td>Optional</td>
<td>A real value</td>
</tr>
</tbody>
</table>

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

```verilog
nature Charge
    abstol = 1e-14 ;
    access = 0 ;
    units = "coul" ;
    blowup = 1e8 ;
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.

```verbatim
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.

```verilog
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.

```verilog
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.

```verilog
discipline neutral
enddiscipline
```

```verilog
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 Figure 4-1 on page 68.

**Figure 4-1 Analog Discipline Compatibility**

Consider the following declarations.
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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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

```plaintext
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]
```

The standard attribute for descriptions can be used with net discipline declarations. For example,

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

Although the `desc` attribute is allowed, Cadence tools, in this release, do nothing with the information.

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.

```plaintext
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.

```plaintext
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.

```verilog
discipline emptydis
dendiscipline
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.

```verilog
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 112.

### 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] vec2[1]` and then compute the difference between the returned value.

```plaintext
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 73.
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 67.

Consider the following declarations.

```verilog
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.

```verilog
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.

```verilog
module diode (a, c) ;
inout a, c ;
electrical a, c ;
branch (a,c) diode, (a,a) anode ; // Declare named branches
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(anode))/$vt) - 1);
    I(diode) <+ white_noise(2* `P_Q * I(diode)) ;
    I(diode) <+ flicker_noise(kf*pow(abs(I(diode)),af),ef);
end
endmodule
```
Statements for the Analog Block

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

- **Procedural Assignment Statements in the Analog Block** on page 76
- **Branch Contribution Statement** on page 76
- **Indirect Branch Assignment Statement** on page 78
- **Sequential Block Statement** on page 79
- **Conditional Statement** on page 80
- **Case Statement** on page 80
- Loop statements, including
  - **Repeat Statement** on page 81
  - **While Statement** on page 82
  - **For Statement** on page 82
- **Generate Statement** on page 83

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.

```
procedural_assignment ::= 
  leexpr = expression ;
leexpr ::= 
  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 ;
```

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

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

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

is equivalent to

```verilog
V(n1, n2) <+ expr1 + expr2 ;
```
Creating a Switch Branch

Important

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 277.

Indirect Branch Assignment Statement

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

\[
\text{indirect\_branch\_assignment} ::= \\
\quad \text{target} : \text{equation} \\
\text{target} ::= \\
\quad \text{bvalue} \\
\text{equation} ::= \\
\quad \text{nexpr} == \text{expression} \\
\text{nexpr} ::= \\
\quad \text{bvalue} \\
\quad \text{ddt} (\text{bvalue}) \\
\quad \text{idt} (\text{bvalue}) \\
\quad \text{idtmod} (\text{bvalue})
\]

An indirect branch assignment has this format:

\[
V(\text{out}) : V(\text{in}) == 0 ;
\]

Read this as “find \( V(\text{out}) \) such that \( V(\text{in}) \) is zero.” This example says that \( \text{out} \) 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 \( V(\text{pin, nin}) \) is zero.”

\[
\text{module opamp (out, pin, nin) ;} \\
\text{output out ;} \\
\text{input pin, nin ;} \\
\text{voltage out, pin, nin ;} \\
\text{analog}
\]
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.

```verilog
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 37.

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.

```verilog
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 47.
Conditional Statement

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

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

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 text_expression matches expression and if you coded a default case_item, the default statement runs. If all 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.

```verilog
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.

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

statement must not include any analog operators. For additional information, see “Analog Operators” on page 143.

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

```verilog
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.

```plaintext
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. For additional information, see “Analog Operators” on page 143.

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

```plaintext
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.

```plaintext
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. For additional information, see “Analog Operators” on page 143.

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.

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

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

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;
  samp = 2.0 * samp;
end 
endmodule
```
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 tool 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 89
- Binary Operators on page 90
- Bitwise Operators on page 93
- Ternary Operator on page 94

For information about precedence and short-circuiting, see:

- Operator Precedence on page 95
- Expression Short-Circuiting on page 95

For information about string operators and functions, see:

- String Operators and Functions on page 95
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 95.

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Definition</th>
<th>Type of Operands Allowed</th>
<th>Example or Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>Unary plus</td>
<td>Integer, real</td>
<td>I = +13; // I = 13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I = +(−13); // I = −13</td>
</tr>
<tr>
<td>−</td>
<td>1</td>
<td>Unary minus</td>
<td>Integer, real</td>
<td>R = −13.1; // R = −13.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I = −(−4−5); // I = 1</td>
</tr>
<tr>
<td>!</td>
<td>1</td>
<td>Logical negation</td>
<td>Integer, real</td>
<td>I = !(1==1); // I = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I = !(1==2); // I = 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>I = !13.2; // I = 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>/<em>Result is zero for a non-zero operand</em>/</td>
</tr>
<tr>
<td>~</td>
<td>1</td>
<td>Bitwise unary negation</td>
<td>Integer</td>
<td>See the Bitwise Unary Negation Operator figure on page 94.</td>
</tr>
<tr>
<td>&amp;</td>
<td>1</td>
<td>Unary reduction AND</td>
<td>integer</td>
<td>See “Unary Reduction Operators.”</td>
</tr>
<tr>
<td>~&amp;</td>
<td>1</td>
<td>Unary reduction NAND</td>
<td>integer</td>
<td>See “Unary Reduction Operators.”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>Unary reduction OR</td>
<td>integer</td>
</tr>
<tr>
<td>~</td>
<td></td>
<td>1</td>
<td>Unary reduction NOR</td>
<td>integer</td>
</tr>
<tr>
<td>^</td>
<td>1</td>
<td>Unary reduction exclusive OR</td>
<td>integer</td>
<td>See “Unary Reduction Operators.”</td>
</tr>
<tr>
<td>^~ or ^</td>
<td></td>
<td>1</td>
<td>Unary reduction exclusive NOR</td>
<td>integer</td>
</tr>
</tbody>
</table>

### 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.
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**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Unary Reduction OR Operator**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Unary Reduction Exclusive OR Operator**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Binary Operators**

The binary operators each require two operands.

**Binary Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Definition</th>
<th>Type of Operands Allowed</th>
<th>Example or Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>3</td>
<td>a plus b</td>
<td>Integer, real</td>
<td>R = 10.0 + 3.1; // R = 13.1</td>
</tr>
</tbody>
</table>
### Binary Operators, continued

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Definition</th>
<th>Type of Operands Allowed</th>
<th>Example or Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>3</td>
<td>(a \text{ minus } b)</td>
<td>Integer, real</td>
<td>(I = 10 - 13; \quad // I = -3)</td>
</tr>
<tr>
<td>*</td>
<td>2</td>
<td>(a \text{ multiplied by } b)</td>
<td>Integer, real</td>
<td>(R = 2.2 * 2.0; \quad // R = 4.4)</td>
</tr>
<tr>
<td>/</td>
<td>2</td>
<td>(a \text{ divided by } b)</td>
<td>Integer, real</td>
<td>(I = 9 / 4; \quad // I = 2) (R = 9.0 / 4; \quad // R = 2.25)</td>
</tr>
<tr>
<td>%</td>
<td>2</td>
<td>(a \text{ modulo } b)</td>
<td>Integer, real</td>
<td>(I = 10 \mod 5; \quad // I = 0) (I = -12 \mod 5; \quad // I = -2) (R = 10 \mod 3.75 \quad // R = 2.5) /<em>The result takes sign of the first operand.</em>/</td>
</tr>
<tr>
<td>&lt;</td>
<td>5</td>
<td>(a \text{ less than } b); evaluates to 0 or 1</td>
<td>Integer, real</td>
<td>(I = 5 &lt; 7; \quad // I = 1) (I = 7 &lt; 5; \quad // I = 0)</td>
</tr>
<tr>
<td>&gt;</td>
<td>5</td>
<td>(a \text{ greater than } b); evaluates to 0 or 1</td>
<td>Integer, real</td>
<td>(I = 5 &gt; 7; \quad // I = 0) (I = 7 &gt; 5; \quad // I = 1)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>5</td>
<td>(a \text{ less than or equal to } b); evaluates to 0 or 1</td>
<td>Integer, real</td>
<td>(I = 5.0 &lt;= 7.5; \quad // I = 1) (I = 5.0 &lt;= 5.0; \quad // I = 1) (I = 5.0 &lt;= 4; \quad // I = 0)</td>
</tr>
<tr>
<td>&gt;=</td>
<td>5</td>
<td>(a \text{ greater than or equal to } b); evaluates to 0 or 1</td>
<td>Integer, real</td>
<td>(I = 5.0 &gt;= 7; \quad // I = 0) (I = 5.0 &gt;= 5; \quad // I = 1) (I = 5.0 &gt;= 4.8; \quad // I = 1)</td>
</tr>
<tr>
<td>==</td>
<td>6</td>
<td>(a \text{ equal to } b); evaluates to 0, 1, or (x) (if any bit of (a) or (b) is (x) or (z)).</td>
<td>Integer, real</td>
<td>(I = 5.2 == 5.2; \quad // I = 1) (I = 5.2 == 5.0; \quad // I = 0) (I = 1 == 1'bx; \quad // I = x)</td>
</tr>
<tr>
<td>!=</td>
<td>6</td>
<td>(a \text{ not equal to } b); evaluates to 0, 1, or (x) (if any bit of (a) or (b) is (x) or (z)).</td>
<td>Integer, real</td>
<td>(I = 5.2 != 5.2; \quad // I = 0) (I = 5.2 != 5.0; \quad // I = 1)</td>
</tr>
</tbody>
</table>
### Binary Operators, continued

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Definition</th>
<th>Type of Operands Allowed</th>
<th>Example or Further Information</th>
</tr>
</thead>
</table>
| &&       | 10         | Logical AND;                      | Integer, real            | I = (1==1) && (2==2); // I = 1  
|          |            | evaluates to 0 or 1               |                          | I = (1==2) && (2==2); // I = 0  
|          |            |                                   |                          | I = -13 && 1; // I = 1                                                 |
| ||       | 11         | Logical OR;                       | Integer, real            | I = (1==2) || (2==2); // I = 1  
|          |            | evaluates to 0 or 1               |                          | I = (1==2) || (2==3); // I = 0  
|          |            |                                   |                          | I = 13 || 0; // I = 1                                                    |
| &        | 7          | Bitwise binary AND                 | Integer                  | See the Bitwise Binary AND Operator figure on page 93.              |
| |        | 9          | Bitwise binary OR                  | Integer                  | See the Bitwise Binary OR Operator figure on page 93.              |
| ^        | 8          | Bitwise binary exclusive OR        | Integer                  | See the Bitwise Binary Exclusive OR Operator figure on page 93.     |
| ^~       | 8          | Bitwise binary exclusive NOR       | Integer                  | See the Bitwise Binary Exclusive NOR Operator figure on page 93.    |
| ~~       | 8          | Bitwise binary exclusive NOR       | Integer                  | See the Bitwise Binary Exclusive NOR Operator figure on page 93.    |
| <<=      | 4          | a shifted b bits left              | Integer                  | I = 1 <<= 2; // I = 4  
|          |            |                                   |                          | I = 2 <<= 2; // I = 8  
|          |            |                                   |                          | I = 4 <<= 2; // I = 16                                                  |
| >>=      | 4          | a shifted b bits right             | Integer                  | I = 4 >> 2; // I = 1  
|          |            |                                   |                          | I = 2 >> 2; // I = 0                                                   |
| or       | 11         | Event OR                           | Event expression         | @(initial_step or cross(V(vin)-1))                                  |
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**

<table>
<thead>
<tr>
<th>&amp;</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bitwise Binary OR Operator**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Bitwise Binary Exclusive OR Operator**

<table>
<thead>
<tr>
<th>^</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Bitwise Binary Exclusive NOR Operator**

<table>
<thead>
<tr>
<th>^~ or ^~</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
<th>Definition</th>
<th>Type of Operands Allowed</th>
<th>Example or Further Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>?:</td>
<td>12</td>
<td>( exp ? t_exp : f_exp )</td>
<td>Valid expressions</td>
<td>( I = 2==3 ? 1:0; ) // ( I = 0 ) ( R = 1==1 ? 1.0:0.0; ) // ( R=1.0 )</td>
</tr>
</tbody>
</table>

A complete conditional operator expression looks like this:

\( \text{conditional} \_\text{expr} \? \text{true} \_\text{expr} : \text{false} \_\text{expr} \)

If \( \text{conditional} \_\text{expr} \) is true, the conditional operator evaluates to \( \text{true} \_\text{expr} \), otherwise to \( \text{false} \_\text{expr} \).

The conditional operator is right associative.

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

\( V(\text{out}) \leftarrow V(\text{in}) > 2.5 \ ? \ 0.0 \ : \ 5.0 \ ; \)

is equivalent to

\begin{verbatim}
If (V(in) > 2.5)
    V(out) <+ 0.0 ;
else
    V(out) <+ 5.0 ;
\end{verbatim}
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.

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+ - ! ~ (unary)</td>
<td>Highest precedence</td>
</tr>
<tr>
<td>2</td>
<td>* /, %</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>+ - (binary)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;&lt;, &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>&lt;=, =&gt;</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>==, !=</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>^, <del>, ^</del></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>?: (conditional operator)</td>
<td>Lowest precedence</td>
</tr>
</tbody>
</table>

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. String operators and functions are supported only in Verilog-A modules that are brought into the design by using the ahdl_include statement.
If you do not need functionality equivalent to SpectreHDL, Cadence recommends using the Verilog-A string functions listed in Table 6-1 on page 96. 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Detailed Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>$sscanf(string_format {,arg})</td>
<td>Reads bytes from a string, interprets the bytes according to the specified string_format format and stores the result in arguments.</td>
<td>“$sscanf” on page 100.</td>
</tr>
<tr>
<td>== != &lt;&gt; &gt;= &lt;=</td>
<td>Compare two strings alphabetically and lexicographically.</td>
<td>“Comparison Operators” on page 99.</td>
</tr>
<tr>
<td>{str_des, str_src}</td>
<td>Appends (concatenates) src_str to des_src.</td>
<td>“Concatenation Operator” on page 99.</td>
</tr>
<tr>
<td>int_as_str.atoi()</td>
<td>Converts a string, int_as_str, to an integer.</td>
<td>“atoi” on page 100.</td>
</tr>
<tr>
<td>real_as_str.atoreal()</td>
<td>Converts a string, real_as_str, to a real.</td>
<td>“atoreal” on page 101.</td>
</tr>
<tr>
<td>str.getc()</td>
<td>Returns the ASCII code of the first character of string1.</td>
<td>“getc” on page 101.</td>
</tr>
<tr>
<td>str.len()</td>
<td>Returns the number of characters in str.</td>
<td>“len” on page 101.</td>
</tr>
<tr>
<td>str.substr(start_pos, end_pos)</td>
<td>Returns the substring of str between start_pos and end_pos, inclusive.</td>
<td>“substr” on page 104.</td>
</tr>
</tbody>
</table>

The SpectreHDL equivalent string functions listed in Table 6-2 on page 97 are provided for those who need functionality equivalent to what was available with the SpectreHDL language. These SpectreHDL equivalent functions are temporary non-standard Cadence extensions.
designed to facilitate the translation of SpectreHDL modules. Support for these functions might be withdrawn in a future release.

The SpectreHDL equivalent string functions must be brought into the design by including the `shdl_strings.vams` file in the module where the functions are used. The `include` statement must be placed after the module declaration but before the `endmodule` statement, just before the `analog` statement.

If you would like to examine the code that implements these functions, the file is provided at
`your_mmsim_install_dir/tools/spectre/etc/ahdl/shdl_strings.vams`

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Detailed Information</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>shdl_strchr(input_str, character)</code></td>
<td>Returns the first position where <code>character</code> is found in <code>input_str</code>.</td>
<td>“shdl_strchr” on page 102.</td>
</tr>
<tr>
<td><code>shdl_strcspn(input_str, span_set)</code></td>
<td>Returns the number of continuous characters from the start of <code>input_str</code> that are not in <code>span_set</code>.</td>
<td>“shdl_strcspn” on page 102.</td>
</tr>
<tr>
<td><code>shdl_strrchr(input_str, character)</code></td>
<td>Returns the last position where <code>character</code> is found in <code>input_str</code>.</td>
<td>“shdl_strrchr” on page 103.</td>
</tr>
<tr>
<td><code>shdl_strspn(input_str, span_set)</code></td>
<td>Returns the number of continuous characters from the start of <code>input_str</code> that are in <code>span_set</code>.</td>
<td>“shdl_strspn” on page 103.</td>
</tr>
<tr>
<td><code>shdl_strstr(input_str, sub_str)</code></td>
<td>Returns the first position where <code>sub_str</code> is found in <code>input_str</code>.</td>
<td>“shdl_strstr” on page 104.</td>
</tr>
</tbody>
</table>
Mapping SpectreHDL String Functions to Verilog-A Functions

The following table provides an overview of the Verilog-A string functions that provide the functionality formerly provided in SpectreHDL.

Table 6-3 Mapping SpectreHDL to Verilog-A String Functions

<table>
<thead>
<tr>
<th>SpectreHDL</th>
<th>Verilog-A</th>
<th>Detailed Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ascii</td>
<td>getc</td>
<td>“getc” on page 101.</td>
</tr>
<tr>
<td>$str</td>
<td>$sscanf</td>
<td>“$sscanf” on page 100.</td>
</tr>
<tr>
<td>$strcat</td>
<td>concatenation operator</td>
<td>“Concatenation Operator” on page 99.</td>
</tr>
<tr>
<td>$strchr</td>
<td>shdl_strchr</td>
<td>“shdl_strchr” on page 102.</td>
</tr>
<tr>
<td>$strcmp</td>
<td>comparison operators</td>
<td>“Comparison Operators” on page 99.</td>
</tr>
<tr>
<td>$strncpy</td>
<td>string copy operator</td>
<td>“String Copy Operator” on page 99.</td>
</tr>
<tr>
<td>$strcspn</td>
<td>shdl_strcspn</td>
<td>“shdl_strcspn” on page 102.</td>
</tr>
<tr>
<td>$strlen</td>
<td>len</td>
<td>“len” on page 101.</td>
</tr>
<tr>
<td>$strchr</td>
<td>shdl_strrchr</td>
<td>“shdl_strrchr” on page 103.</td>
</tr>
<tr>
<td>$strspn</td>
<td>shdl_strspn</td>
<td>“shdl_strspn” on page 103.</td>
</tr>
<tr>
<td>$strstr</td>
<td>shdl_strstr</td>
<td>“shdl_strstr” on page 104.</td>
</tr>
<tr>
<td>$strtoint</td>
<td>atoi</td>
<td>“atoi” on page 100.</td>
</tr>
<tr>
<td>$strtoreal</td>
<td>atoreal</td>
<td>“atoreal” on page 101.</td>
</tr>
<tr>
<td>$substr</td>
<td>substr</td>
<td>“substr” on page 104.</td>
</tr>
</tbody>
</table>

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.

\[
\text{comparison_operator ::= str1 \{ \text{==, !=, <, <=, >, >=} \str2 \}}
\]

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,

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

String Copy Operator

Use the string copy operator to copy a string.

\[
\text{string_copy_operator ::= str2 \text{=} str1}
\]

For example,

```vlog
des_str = src_str;
```

copies src_str to des_str.

Concatenation Operator

Use the concatenation operator to append (concatenate) a string to another string.

\[
\text{string_concatenation_operator ::= \{ str1, str2 \}}
\]

For example,

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

appends (concatenates) str_src to str_des.
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 create a string from mixed-type arguments. It works like $strobe(). See "$strobe" on page 166 for more information.

```
$sscanf_function ::= $sscanf(string_format, arg)
```

For example, the following function creates a string from an integer, a string, and a real variable.

```vhdl
integer varInt;
real varReal;
string varString;
string retString;
@(initial_step)
begin
    varInt = 123;
    varString = "456";
    varReal = 7.890121212e2;
    retString = $sscanf("Use Integer %d, string %s and real %.1f to create a string %d%s%.1f !", varInt, varString, varReal, varInt, varString, varReal);
end
```

For this example, retString receives the value "Use Integer 123, string 456 and real 789.0 to create a string 123456789.0!"

atoi

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

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

For example,

```vhdl
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

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.

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=.cumeg", ";" ); // returns 6
```
shdl_strrchr

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

\[
\text{shdl_strrchr} \text{ function ::=}
\begin{align*}
\text{shdl_strrchr} & \text{ (input_string, character)}
\end{align*}
\]

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.

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("abcdefg\2", "1"); // Returns -1
```

shdl_strspn

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

\[
\text{shdl_strspn} \text{ function ::=}
\begin{align*}
\text{shdl_strspn} & \text{ (input_string, span_set)}
\end{align*}
\]

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.

For example:
```
'include "shdl_strings.vams"
...
p1 = shdl_strstr("a little string in a big string", "little"); // Returns 2
p2 = 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. Notice that this is different from the SpectreHDL $substr, which does not include the end_pos character in the substring. 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 106
- Trigonometric and Hyperbolic Functions on page 106
- Controlling How Math Domain Errors Are Handled on page 107

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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Domain</th>
<th>Returned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>Absolute</td>
<td>All (x)</td>
<td>Integer, if (x) is integer; otherwise, real</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>Smallest integer larger than or equal to (x)</td>
<td>All (x)</td>
<td>Integer</td>
</tr>
<tr>
<td>exp(x)</td>
<td>Exponential. See also &quot;Limited Exponential Function&quot; on page 144.</td>
<td></td>
<td>Real</td>
</tr>
<tr>
<td>floor(x)</td>
<td>Largest integer less than or equal to (x)</td>
<td>All (x)</td>
<td>Integer</td>
</tr>
<tr>
<td>ln(x)</td>
<td>Natural logarithm</td>
<td>(x &gt; 0)</td>
<td>Real</td>
</tr>
<tr>
<td>log(x)</td>
<td>Decimal logarithm</td>
<td>(x &gt; 0)</td>
<td>Real</td>
</tr>
<tr>
<td>max((x, y))</td>
<td>Maximum</td>
<td>All (x, y)</td>
<td>Integer, if (x) and (y) are integers; otherwise, real</td>
</tr>
<tr>
<td>min((x, y))</td>
<td>Minimum</td>
<td>All (x, y)</td>
<td>Integer, if (x) and (y) are integers; otherwise, real</td>
</tr>
<tr>
<td>pow((x, y))</td>
<td>Power of ((x^y))</td>
<td>All (y, y \geq 0) (y &gt; 0), if (x = 0) (y) integer, if (x &lt; 0)</td>
<td>Real</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>Square root</td>
<td>(x \geq 0)</td>
<td>Real</td>
</tr>
</tbody>
</table>

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.

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

**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.

The `ahdldomainerror` parameter format is

```
Name options ahdldomainerror=value
```

where the syntax items are defined as follows.
For example, you might have the following in a Spectre control file to ensure that simulation stops when a domain error occurs.

```plaintext
myoption options ahdldomainerror=error
```
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 111
- Final_step Event on page 111
- Cross Event on page 112
- Above Event on page 113
- Timer Event on page 115
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

<table>
<thead>
<tr>
<th>initial_step_event</th>
<th>“Initial_step Event” on page 111</th>
</tr>
</thead>
<tbody>
<tr>
<td>final_step_event</td>
<td>“Final_step Event” on page 111</td>
</tr>
<tr>
<td>cross_event</td>
<td>“Cross Event” on page 112</td>
</tr>
<tr>
<td>above_event</td>
<td>“Above Event” on page 113</td>
</tr>
</tbody>
</table>
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
results at the end of the analysis. (This example also uses the timer event, which is discussed in “Timer Event” on page 115.)

```verilog
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 ] ] ])

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.

<table>
<thead>
<tr>
<th>If you want to</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect all zero crossings</td>
<td>Do not specify <code>direction</code>, or set <code>direction</code> equal to 0</td>
</tr>
<tr>
<td>Detect only zero crossings where the value is increasing</td>
<td>Set <code>direction</code> equal to +1</td>
</tr>
</tbody>
</table>
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

\[ 1e-9 + \text{reltol} * \text{max_value_of_the_signal} \]

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

The cross function is subject to the restrictions listed in “Restrictions on Using Analog Operators” on page 144.

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.

```verilog
dmodule 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 ] ] )
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.

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.

The above function is subject to the restrictions listed in “Restrictions on Using Analog Operators” on page 144.

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
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 ::= \\
\text{timer ( start_time \[,\ period \[,\ timetol \])}
\]

\(start_time\) 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.

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

\[
\text{module squarewave (out)} \\
\text{output out ;} \\
\text{electrical out ;} \\
\text{parameter period = 1.0 ;} \\
\text{integer x ;} \\
\text{analog begin} \\
\quad @(\text{initial_step}) x = 1 ; \\
\quad @(\text{timer(0, period/2)}) x = -x ; \\
\quad V(out) <+ \text{transition}(x, 0.0, \text{period}/100.0) ; \\
\text{end} \\
\text{endmodule}
\]
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 119
- Bounding the Time Step on page 121
- Finding When a Signal Is Zero on page 122
- Querying the Simulation Environment on page 123
- Detecting Parameter Overrides on page 125
- Obtaining and Setting Signal Values on page 126
- Determining the Current Analysis Type on page 128
- Implementing Small-Signal AC Sources on page 130
- Implementing Small-Signal Noise Sources on page 130
- Generating Random Numbers on page 132
- Generating Random Numbers in Specified Distributions on page 133
- Interpolating with Table Models on page 139

For information on analog operators and filters, see

- Limited Exponential Function on page 144
- Time Derivative Operator on page 144
- Time Integral Operator on page 145
- Circular Integrator Operator on page 147
Delay Operator on page 150
Transition Filter on page 151
Slew Filter on page 154
Implementing Laplace Transform S-Domain Filters on page 156
Implementing Z-Transform Filters on page 161

For descriptions of functions used to control input and output, see
Displaying Results on page 165
Working with Files on page 171

For descriptions of functions used to control the simulator, see
Exiting to the Operating System on page 176

For a description of the \$pwr function, which is used to specify power consumption in a module, see
Specifying Power Consumption on page 170

For information on using user-defined functions in the Verilog-A language, see
Declaring an Analog User-Defined Function on page 177
Calling a User-Defined Analog Function on page 179
Announcing Discontinuity

Use the $\text{discontinuity}$ function to tell the simulator about a discontinuity in signal behavior.

\[
\text{discontinuity\_function} ::= \$\text{discontinuity}[\ (\text{constant\_expression}) ]
\]

$\text{constant\_expression}$, which must be zero or a positive integer, is the degree of the discontinuity. For example, $\text{discontinuity}$, which is equivalent to $\text{discontinuity}(0)$, indicates a discontinuity in the equation, and $\text{discontinuity}(1)$ 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 $\text{transition}$ and $\text{slew}$.

Be aware that using the $\text{discontinuity}$ 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 $\text{discontinuity}$ function while describing the behavior of a source that generates a triangular wave. As the Triangular Wave figure on page 119 shows, the triangular wave is continuous, but as the Triangular Wave First Derivative figure on page 119 shows, the first derivative of the wave is discontinuous.

**Triangular Wave**

![Triangular Wave](image)

**Triangular Wave First Derivative**

![Triangular Wave First Derivative](image)

The module $\text{trisource}$ describes this triangular wave source.

\[
\text{module trisource (vout)};
\text{output vout;}
\text{voltage vout;}
\text{parameter real wavelength = 10.0, amplitude = 1.0;}
\text{integer slope;}
\text{real wstart;}
\]
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 120 through module relay produces the discontinuous current shown in the Relay Current figure on page 121.

**Relay Voltage**

![Relay Voltage Graph](image)
Bounding the Time Step

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

\[
\text{bound_step} \text{ function} ::= \\
\quad \text{$bound_step ( \text{max_step} )}$
\]

\[
\text{max_step} ::= \\
\quad \text{constant_expression}
\]

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

\[
\begin{align*}
\text{module sinwave (outsig) ;} \\
\text{output outsig ;} \\
\text{voltage outsig ;} \\
\text{parameter real freq = 1.0, ampl = 1.0 ;} \\
\text{analog begin} \\
\quad \text{V(outsig) <+ ampl * sin(2.0 * 'M_PI * freq * $abstime) ;} \\
\quad \text{$bound_step(0.02 / freq) ;}$ \quad \text{// Max time step = 1/50 period} \\
\text{end}
\end{align*}
\]

Announcing and Handling Nonlinearities

Use the $limit function to announce nonlinearities that are other than exponential. This information is used to improve convergence.

\[
\text{limit_call_function} ::= \\
\quad \text{$limit ( \text{access_function_reference} )}$
\quad \text{||} \\
\quad \text{$limit ( \text{access_function_reference, string, arg_list} )}$
\quad \text{||} \\
\quad \text{$limit ( \text{access_function_reference, analog_function_ID, arg_list} )$
\]

\[
\text{access_function_reference} \text{ is the reference that is being limited.}
\]

\[
\text{string} \text{ is a built-in simulator function that you recommend be used to compute the return value. In this release, the syntax of string is not checked.}
\]
analog_function_ID is a user-defined analog function that you recommend be used to compute the return value. In this release, the syntax of analog_function_ID is not checked.

arg_list is a list of arguments for the built-in or user-defined function. In this release, the syntax of arg_list is not checked.

**Note:** Although the $limit function is allowed, Cadence tools, in this release, do nothing with the information. Consequently, coding

```
vdio = $limit(V(a,c), spicepnjlim, $vt, vcrit);
```

is equivalent to coding

```
vdio = V(a,c);
```

## 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.

<table>
<thead>
<tr>
<th>If you want to</th>
<th>Then</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detect all crossings</td>
<td>Set direction equal to 0</td>
</tr>
<tr>
<td>Detect only crossings where the value is increasing</td>
<td>Set direction equal to +1</td>
</tr>
<tr>
<td>Detect only crossings where the value is decreasing</td>
<td>Set direction equal to -1</td>
</tr>
</tbody>
</table>

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 144.

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.

```verilog
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
endmodule
```

### Querying the Simulation Environment

Use the simulation environment functions described in the following sections to obtain information about the current simulation environment.

#### Obtaining the Current Simulation Time

Verilog-A provide two environment parameter functions that you can use to obtain the current simulation time: `$abstime` and `$realtime`.

**$abstime Function**

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

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

**$realtime Function**

Use the `$realtime` function to obtain the current simulation time in seconds.
realtime_function ::=  
   $realtime[time_scale]

$time_scale$ is a value used to scale the returned simulation time. The valid values are the integers 1, 10, and 100, followed by one of the scale factors in the following table.

<table>
<thead>
<tr>
<th>Scale Factor</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>Seconds</td>
</tr>
<tr>
<td>ms</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>us</td>
<td>Microseconds</td>
</tr>
<tr>
<td>ns</td>
<td>Nanoseconds</td>
</tr>
<tr>
<td>ps</td>
<td>Picoseconds</td>
</tr>
<tr>
<td>fs</td>
<td>Femtoseconds</td>
</tr>
</tbody>
</table>

If you do not specify $time_scale$, the return value is scaled to the `$time_unit` of the module that invokes the function.

For example, to print out the current simulation time in seconds, you might code

$s strobe("Simulation time = %e", $realtime(1s)) ;$

**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.

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

`param` is one of the following simulation parameters.

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

`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

```verbatim
$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

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

## 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.

```verbatim
$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.

The `$param_given` function can be used 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.

```verbatim
```
if ($param_given(tdevice))
    temp = tdevice + 'P_CELSIUS0;
else
    temp = $temperature;

**Obtaining and Setting Signal Values**

Use the access functions to obtain or set the signal values.

```verbatim
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_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 equal to `V`. Consequently, the access function for electrical potential is named `V`. For additional information, see Appendix C, “Standard Definitions.”

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

<table>
<thead>
<tr>
<th>Format</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>( V(b1) )</td>
<td>Accesses the potential across branch ( b1 )</td>
</tr>
<tr>
<td>( V(n1) )</td>
<td>Accesses the potential of ( n1 ) relative to ground</td>
</tr>
<tr>
<td>( V(n1,n2) )</td>
<td>Accesses the potential difference on the unnamed branch between ( n1 ) and ( n2 )</td>
</tr>
<tr>
<td>( I(b1) )</td>
<td>Accesses the current on branch ( b1 )</td>
</tr>
<tr>
<td>( I(n1) )</td>
<td>Accesses the current flowing from ( n1 ) to ground</td>
</tr>
<tr>
<td>( I(n1, n2) )</td>
<td>Accesses the current flowing on the unnamed branch between ( n1 ) and ( n2 ); node ( n1 ) and node ( n2 ) cannot be the same node</td>
</tr>
<tr>
<td>( I(p1,p1) )</td>
<td>Accesses the current flow into the module through port ( p1 ). This format accesses the port branch associated with port ( p1 ).</td>
</tr>
</tbody>
</table>

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.

```verilog
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,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
```

December 2006 127 Product Version 6.1
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.

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

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Analysis Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>AC analysis</td>
</tr>
<tr>
<td>dc</td>
<td>OP or DC analysis</td>
</tr>
<tr>
<td>dcmatch</td>
<td>DC device matching analysis</td>
</tr>
<tr>
<td>montecarlo</td>
<td>Monte Carlo analysis</td>
</tr>
<tr>
<td>pac</td>
<td>Periodic AC (PAC) analysis</td>
</tr>
<tr>
<td>pnoise</td>
<td>Periodic noise (PNoise) analysis</td>
</tr>
</tbody>
</table>
### Analysis Types and Descriptions, continued

<table>
<thead>
<tr>
<th>Analysis Type</th>
<th>Analysis Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pss</td>
<td>Periodic steady-state analysis</td>
</tr>
<tr>
<td>pxf</td>
<td>Periodic transfer function analysis</td>
</tr>
<tr>
<td>qpss</td>
<td>Quasi-periodic steady state analysis</td>
</tr>
<tr>
<td>sp</td>
<td>S-parameter analysis</td>
</tr>
<tr>
<td>static</td>
<td>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</td>
</tr>
<tr>
<td>stb</td>
<td>Stability analysis</td>
</tr>
<tr>
<td>tdr</td>
<td>Time-domain reflectometer analysis</td>
</tr>
<tr>
<td>tran</td>
<td>Transient analysis</td>
</tr>
<tr>
<td>xf</td>
<td>Transfer function analysis</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Argument</th>
<th>DC</th>
<th>TRAN OP</th>
<th>TRAN TRAN</th>
<th>AC OP</th>
<th>AC</th>
<th>NOISE OP</th>
<th>NOISE AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>ic</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>dc</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tran</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ac</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>noise</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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.

```verilog
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 128. 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.

```verilog
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:

```verilog
I(diode) <+ white_noise(2 * 'P_Q * Id, "shot") ;
```

For a resistor, you might use a fragment like the following.

```verilog
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:

\[ \frac{1}{f^{\exp}} \]

The syntax for the `flicker_noise` function is

```verilog
flicker_noise( power, exp [ , "name"])
```

`power` is the power of the source at 1 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 `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:

```verilog
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.

```verilog
noise_table(vector [ , "name"])
```

`vector` is an array containing pairs of real numbers. The first number in each pair is a frequency in hertz; the second number is the power at that frequency. The `noise_table` function uses linear interpolation to compute the spectral density for each frequency. At frequencies lower than the lowest frequency specified in the table, the associated power is
assumed to be the power associated with the lowest specified frequency. Similarly, at frequencies higher than the highest frequency specified in the table, the associated power is assumed to be the power associated with the highest specified frequency.

`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:

```verilog
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` function to generate a signed integer, 32-bit, pseudorandom number.

```verilog
$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.

```verilog
// 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
```
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.

```verilog
$rdist_uniform (seed, start, end);
$dist_uniform (seed, start, end);
```

`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 $start$.

The following module returns a series of real numbers, each of which is between 20 and 60 inclusively.

```verilog
defmodule 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 ) ;
    $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.

```verilog
$rdist_normal ( seed , mean , standard_deviation ) ;
$dist_normal ( seed , mean , standard_deviation ) ;
```

$seed$ 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.
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.

```verilog
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.

```verilog
$rdist_exponential ( seed , mean ) ;
$dist_exponential ( seed , mean ) ;
```

`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.

For example, the following module returns a series of real numbers that together form an exponential distribution.

```verilog
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_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);
$dist_poisson (seed, mean);

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.

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);
$dist_chi_square (seed, degree_of_freedom);

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.

For example, the following module returns a series of real numbers that together form a chi-square distribution.

```verilog
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_chisquare(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.

```verilog
$rdist_t ( seed , degree_of_freedom ) ;
$dist_t ( seed , degree_of_freedom ) ;
```

*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.

For example, the following module returns a series of real numbers that together form a Student's T distribution.

```verilog
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
```
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 ) ;
$dist_erlang ( seed , k , mean ) ;
```

`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.

`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.

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
    rrandnum = $rdist_erlang(seed, k, mean) ;
    $display ("Random number is %g", rrandnum ) ;
    V(pinout) <+ rrandnum ;
  end /* of analog block */
endmodule
```
Interpolating with Table Models

Use the $table_model function to model the behavior of a design by interpolating between and extrapolating outside of data points.

$$table_model_declaration ::= \$table_model( variables , \text{table_source} [ , \text{ctrl_string} ] )$$

$$variables ::= \text{independent_var} \{ , \text{independent_var} \}$$

$$table_source ::= \text{data_file}$$

$$\text{data_file} ::= \"filename\"$$

$$\text{table_model_array} ::= \text{array_ID} \{ , \text{array_ID}, \text{output_array_ID} \}$$

$$\text{ctrl_string} ::= \"\text{sub_ctrl_string} \{ , \text{sub_ctrl_string} \}\"$$

$$\text{sub_ctrl_string} ::= \text{I}$$

$$\text{I} ::= [ \text{degree_char} ] \{ \text{extrap_char} [ \text{extrap_char} ] \}$$

$$\text{degree_char} ::= 1 | 2 | 3$$

$$\text{extrap_char} ::= C | L | S | E$$

$independent_var$ is a numerical expression used as an independent model variable. It can be any legal expression that can be assigned to an analog signal. There must be an independent model variable specified for each dimension with a corresponding $\text{sub_ctrl_string}$ other than $\text{I}$ (ignore). There must not be an independent model variable specified for dimensions that have a $\text{sub_ctrl_string}$ of $\text{I}$ (ignore).

data_file is the text file that stores the sample points. You can either give the file name directly or use a string parameter. For more information, see “Table Model File Format” on page 140.

table_model_array is a set of one-dimensional arrays that contains the data points to be passed to the $table_model function. The size of the arrays is the same as the number of sample points. The data is stored in the arrays so that for the $k^{th}$ dimension of the $i^{th}$ sample point, $k^{th\_dim\_array\_identifier}[i] = X_{ik}$ and so that for the $i^{th}$ sample point $output\_array\_identifier[i] = Y_i$. For an example, see “Example: Preparing Data in One-Dimensional Array Format” on page 142.

$\text{ctrl_string}$ controls the numerical aspects of the interpolation process. It consists of subcontrol strings for each dimension.
sub_ctrl_string specifies the handling for each dimension.

The I (ignore) value specifies that the corresponding dimension (column) in the data file is to be ignored. 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.

The D (discrete) value specifies that no interpolation is to occur for this dimension. If the exact value passed to the function for the dimension is not found in the corresponding dimension in the data file, an error is issued and simulation stops.

degree_char is the degree of the splines used for interpolation. The degree must not be zero or exceed 3. 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 (clamp) extrapolation method uses a horizontal line that passes through the nearest sample point, also called the end point, to extend the model evaluation. The L (linear) extrapolation method, which is the default method, models the extrapolation through a tangent line at the end point. The S (spline) extrapolation method uses the polynomial for the nearest segment (the segment at the end) to evaluate a point beyond the interpolation area. The E (error) extrapolation method issues a warning when the point to be evaluated is 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 144.

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 \textit{fastest changing} independent variable and the base current is the \textit{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:

\[ P_1 \\
P_2 \\
P_3 \\
\ldots \\
P_M \]

where \( P_i \) \((i = 1...M)\) are the sample points. Each sample point \( P_i \) is on a separate line and is represented as a sequence of numbers, \( X_{i1} X_{i2} \ldots X_{iN} Y_i \) 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 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.

\textbf{Isoline 1:} \( x=1 \)
\[ y = 1, 2, 3, 4 \\
z = 2, 5, 10, 17 \]

\textbf{Isoline 2:} \( x=2 \)
\[ y = 1, 2, 3, 4 \\
z = 3, 6, 11, 18 \]

\textbf{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.

```verilog
`include "disciplines.vams"
`include "constants.vams"
module mynmos (g, d, s);
electrical g, d, s;
inout g, d, s;
analog begin
    I(d, s) <+ $table_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 $f_{xy}$.

```verilog
module measured_resistance (a, b);
electrical a, b;
```
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.

\[ \text{limexp}( \text{expr} ) \]

\( \text{expr} \) 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 106.

The limexp function is subject to the restrictions listed in “Restrictions on Using Analog Operators” on page 144.

Time Derivative Operator

Use the time derivative operator to calculate the time derivative of an argument.

\[ \text{ddt}( \text{input} [ , \text{abstol} | \text{nature} ] ) \]

\( \text{input} \) is a dynamic expression.

\( \text{abstol} \) is a constant specifying the absolute tolerance that applies to the output of the ddt operator. Set \( \text{abstol} \) at the largest signal level that you consider negligible. In this release of Verilog-A, \( \text{abstol} \) is ignored.

\( \text{nature} \) is a nature from which the absolute tolerance is to be derived. In this release of Verilog-A, \( \text{nature} \) is ignored.
The time derivative operator is subject to the restrictions listed in “Restrictions on Using Analog Operators” on page 144.

In DC analyses, the \( ddt \) operator returns 0. In small-signal analyses, the \( ddt \) operator phase-shifts \( expr \) according to the following formula.

\[
output(\omega) = j \cdot \omega \cdot input(\omega)
\]

To define a higher order derivative, you must use an internal node or signal. For example, a statement such as the following is illegal.

\[
V(out) \leftarrow ddt(ddt(V(in))) \ // \text{ILLEGAL!}
\]

For an example illustrating how to define higher order derivatives correctly, see “Using Integration and Differentiation with Analog Signals” on page 40.

**Time Integral Operator**

Use the time integral operator to calculate the time integral of an argument.

\[
idt( \ input \ [ , \ ic \ [ , \ assert \ [ , \ abstol \ | \ nature \ ] ] ] )
\]

*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 144.
The value returned by the `idt` operator during DC or AC analysis depends on which of the parameters you specify.

<table>
<thead>
<tr>
<th>If you specify</th>
<th>Then idt returns</th>
</tr>
</thead>
</table>
| `input`        | \[ \int_0^t x(\tau) \, d\tau \]  
The time-integral of \( x \) from 0 to \( t \) with the initial condition being computed in the DC analysis. |
| `input, ic`    | \[ \int_0^t x(\tau) \, d\tau + ic \]  
The time-integral of \( x \) from 0 to \( t \) with initial condition \( ic \). In DC or IC analyses, returns \( ic \). |
| `input, ic, assert` | \[ \int_{t_0}^t x(\tau) \, d\tau + ic \]  
The time-integral of \( x \) from \( t_0 \) to \( t \) with initial condition \( ic \). In DC or IC analyses, and when `assert` is nonzero, returns \( ic \). \( t_0 \) is the time when `assert` last became 0. |
| `input, ic, assert, abstol` | \[ \int_{t_0}^t x(\tau) \, d\tau + ic \]  
The time-integral of \( x \) from \( t_0 \) to \( t \) with initial condition \( ic \). In DC or IC analysis, and when `assert` is nonzero, returns \( ic \). \( t_0 \) is the time when `assert` last became 0. |
| `input, ic, assert, nature` | \[ \int_{t_0}^t x(\tau) \, d\tau + ic \]  
The time-integral of \( x \) from \( t_0 \) to \( t \) with initial condition \( ic \). In DC or IC analysis, and when `assert` is nonzero, returns \( ic \). \( t_0 \) is the time when `assert` last became 0. |

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.

\[ V(\text{out}) \leftarrow \sin(2 \cdot \pi \cdot (f_c \cdot \text{abstime} + \text{idt}(\text{gain} \cdot V(\text{in}), 0))) ; \]

**Circular Integrator Operator**

Use the circular integrator operator to convert an expression argument into its indefinitely integrated form.

\[
\text{idtmod}(\text{expr} [, \text{ic} [, \text{modulus} [, \text{offset} [, \text{abstol} | \text{nature} ] ] ] ])
\]

- `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

\[
\text{offset} < \text{idtmod}\_\text{output} < \text{offset} + \text{modulus}
\]

`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.

The circular integrator operator is subject to the restrictions listed in “Restrictions on Using Analog Operators” on page 144.

The value returned by the `idtmod` operator depends on which parameters you specify.

<table>
<thead>
<tr>
<th>If you specify</th>
<th>Then <code>idtmod</code> returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>expr</code></td>
<td>[ x = \int_{0}^{t} \text{expr}(\tau) d\tau ]</td>
</tr>
<tr>
<td></td>
<td>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>.</td>
</tr>
</tbody>
</table>
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 [0,2π] 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 144.

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 \texttt{absdelay} operator to delay the entire signal of a continuously valued waveform.

\texttt{absdelay( expr, time\_delay [, max\_delay ] )}

\textit{expr} is a dynamic expression to be delayed.

\textit{time\_delay}, a dynamic nonnegative value, is the length of the delay. If you specify \textit{max\_delay}, you can change the value of \textit{time\_delay} during a simulation, as long as the value remains in the range \(0 < \textit{time\_delay} < \textit{max\_delay}\). Typically \textit{time\_delay} is a constant but can also vary with time (when \textit{max\_delay} is defined).

\textit{max\_delay} is a constant nonnegative number greater than or equal to \textit{time\_delay}. You cannot change \textit{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

\texttt{V(out) <+ absdelay(V(in), 5u) ;}

The \texttt{absdelay} operator is subject to the restrictions listed in “Restrictions on Using Analog Operators” on page 144.

In DC and operating analyses, the \texttt{absdelay} operator returns the value of \textit{expr} unchanged. In small-signal analyses, the \texttt{absdelay} operator phase-shifts \textit{expr} according to the following formula.

\[
output(\omega) = input(\omega) \cdot e^{-j\omega \cdot time\_delay}
\]

In time-domain analyses, the \texttt{absdelay} operator introduces a transport delay equal to the instantaneous value of \textit{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.

\[ \text{transition}(\text{input} [, \text{delay} [, \text{rise\_time} [, \text{fall\_time} [, \text{time\_tol} ]]]]) \]

\text{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 \text{input}. Changes in \text{input} do not have an effect on the output value until \text{delay} seconds have passed.

\text{delay} is a dynamic nonnegative real value that is an initial delay. By default, \text{delay} has a value of zero.

\text{rise\_time} is a dynamic positive real value specifying the time over which you want positive transitions to occur. If you do not specify \text{rise\_time} or if you give \text{rise\_time} a value of 0, \text{rise\_time} defaults to the value defined by `default\_transition`.

\text{fall\_time} is a dynamic positive real number specifying the time over which you want negative transitions to occur. By default, \text{fall\_time} has the same value that \text{rise\_time} has. If you do not specify \text{rise\_time} or if you give \text{rise\_time} a value of 0, \text{fall\_time} defaults to the value defined by `default\_transition`.

\text{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 \text{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 144.

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.

![Diagram showing the effect of a transition filter on a waveform](image)

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.

```verilog
// comparator model
analog begin
    if ( V(in) > 0 ) begin
        Vout = 5;
    end
    else begin
        Vout = 0;
    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 154.
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.

<table>
<thead>
<tr>
<th>If the new destination value is below the value at the point of interruption, the transition filter</th>
<th>If the new destination value is above the value at the point of interruption, the transition filter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Uses the value of the original destination as the value of the new origin.</td>
<td>1. Retains the original origin.</td>
</tr>
<tr>
<td>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 fall_time seconds.</td>
<td>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 rise_time seconds.</td>
</tr>
<tr>
<td>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.</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

```verilog
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. 

```verilog
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.

```verilog
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 151 for more information.

```verilog
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*. 

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 144.

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.

\[
\frac{\Delta y}{\Delta t} \leq \text{max_pos_rate}
\]

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 144. 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:

```verilog
class 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.

\[ \text{laplace_zp}(expr, \zeta, \rho[, \epsilon]) \]

\( \zeta \) (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

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

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.
\textbf{laplace\_zd(}expr, \zeta, d[ , \varepsilon ])\textbf{)

}\zeta (\text{zeta}) \text{ is a fixed-sized vector of } M \text{ 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 \text{ is a fixed-sized vector of } N \text{ real numbers that contains the coefficients of the denominator. If you use array literals to define the } \zeta \text{ and } d \text{ 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 \( (1 - s/\zeta) \).

\textbf{Numerator-Pole Laplace Transforms}

Use \textbf{laplace\_np} to implement the numerator-pole form of the Laplace transform filter.

\textbf{laplace\_np(}expr, \ n, \rho[ , \varepsilon ])\textbf{)

}\( n \) \text{ is a fixed-sized vector of } M \text{ real numbers that contains the coefficients of the numerator. } \rho \text{ (rho) is a fixed-sized vector of } N \text{ 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 \text{ and } \rho \text{ 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 \( k^{th} \) power of \( s \) in the numerator, and \( \rho_k^r \) and \( \rho_k^i \) are the real and imaginary parts of the \( k^{th} \) 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.

\[
\text{laplace\_nd} (expr, n, d[, \varepsilon])
\]

\( n \) 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 \( k^{th} \) power of \( s \) in the numerator, and \( d_k \) is the coefficient of the \( k^{th} \) power of \( s \) in the denominator.

### Examples

The following code fragments illustrate how to use the Laplace transform filters.

\[
V(\text{out}) \leftarrow \text{laplace\_zp}(V(\text{in}), (0,0), \{1,2,1,-2\});
\]

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

\[
V(\text{out}) \leftarrow \text{laplace\_nd}(V(\text{in}), (0,1), \{1,-0.4,0.2\});
\]
is equivalent.

The following statement contains an empty vector:

\[ V(\text{out}) <+ \text{laplace}_z\text{p}(V(\text{in}), (), (-1,0)); \]

The absence of zeros, indicated by the empty brackets, 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) <+ \text{laplace}_z\text{d}(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
  @(\text{initial}_\text{step}("\text{static}")) begin
    nn[0] = 1 ;  // These assignment
    nn[1] = 2 ;  // statements run only
    dd[0] = 1 ;  // at the beginning of
    dd[1] = 6 ;  // the analyses.
  end
  V(pout, nout) <+ \text{laplace}_n\text{d}(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.
`\text{include } "\text{constants.vams}\"
`\text{include } "\text{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 144.

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 \( k^{th} \) 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 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, ζ, d, T [, τ [, t0] ])
```

ζ (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 ζ 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

\[
H(z) = \frac{M - 1}{N - 1} \prod_{k=0}^{M-1} \left(1 - z^{-1}(\xi_k^r + j\xi_k^i)\right) \sum_{k=0}^{N-1} d_k z^{-k}
\]

where \(\xi_k^r\) and \(\xi_k^i\) are the real and imaginary parts of the \(k^{th}\) zero, and \(d_k\) is the coefficient of the \(k^{th}\) power of \(z\) 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 \((1 - (z^{-1} \cdot \xi))\).

Numerator-Pole Z-Transforms

Use `zi_np` to implement the numerator-pole form of the Z-transform filter.

```
zi_np(expr, n, ρ, T [, τ [, t0] ])
```

n is a fixed or parameter-sized vector of M real numbers that contains the coefficients of the numerator. ρ (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 ρ vectors, the values must be constant or dependent upon parameters only. You cannot use array values defined by variables.

The transfer function is
where \( n_k \) is the coefficient of the \( k^{th} \) power of \( z \) in the numerator, and \( \rho_k^r \) and \( \rho_k^i \) are the real and imaginary parts of the \( k^{th} \) 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 \( (1 - z^{-1}) \).

### Numerator-Denominator Z-Transforms

Use `zi_nd` to implement the numerator-denominator form of the Z-transform filter.

\[
zi\_nd(expr, \ n, \ d, \ T [ , \ t [ , \ t_0] ]) \]

\( n \) 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 \( k^{th} \) power of \( z \) in the numerator, and \( d_k \) is the coefficient of the \( k^{th} \) power of \( s \) in the denominator.

### Examples

The following example illustrates an ideal sampled data integrator with the transfer function

\[
Hz(n k z k - k = M - 1 \sum_{k=0}^{N-1} d_k z^{-k} = n k z^{-k}) \]

\[
\prod_{k=0}^{N-1} \left(1 - z^{-1}(\rho_k^r + j\rho_k^i)\right) \]

\[
= \frac{1}{\prod_{k=0}^{N-1} \left(1 - z^{-1}(\rho_k^r + j\rho_k^i)\right)} \]

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

This transfer function can be implemented as

```verilog
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.

```verilog
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 ::= \\
\quad \text{$strobe} \[ \{ \text{list_of_arguments} \} \]
\]
\[
\text{list_of_arguments ::=} \\
\quad \text{argument} \\
\quad | \quad \text{list_of_arguments, argument}
\]

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.

<table>
<thead>
<tr>
<th>Use this sequence</th>
<th>To include</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>The new-line character</td>
</tr>
<tr>
<td>\t</td>
<td>The tab character</td>
</tr>
<tr>
<td>&quot;</td>
<td>The backslash character, \</td>
</tr>
<tr>
<td>&quot;</td>
<td>The quotation mark character, &quot;</td>
</tr>
<tr>
<td>\ddd</td>
<td>A character specified by 1 to 3 octal digits</td>
</tr>
<tr>
<td>%</td>
<td>The percent character, %</td>
</tr>
<tr>
<td>%m or %M</td>
<td>The hierarchical name of the current module, function, or named block</td>
</tr>
</tbody>
</table>
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:

<table>
<thead>
<tr>
<th>flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>–</td>
<td>Left justify the output</td>
</tr>
<tr>
<td>+</td>
<td>Always print a sign</td>
</tr>
<tr>
<td>Blank space, or any character other than a sign</td>
<td>Print a space</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>format_character</th>
<th>Type of Argument</th>
<th>Output</th>
<th>Example Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>b or B</td>
<td>Binary</td>
<td>Binary format</td>
<td>00000000000000000000000000000000000011100</td>
</tr>
<tr>
<td>c or C</td>
<td>Integer</td>
<td>ASCII character format</td>
<td>191, 48, -567</td>
</tr>
<tr>
<td>d or D</td>
<td>Integer</td>
<td>Decimal format</td>
<td>-1.0, 4e8, 34.349e-12</td>
</tr>
<tr>
<td>e or E</td>
<td>Real</td>
<td>Real, exponential format</td>
<td>-1.0, 4e8, 34.349e-12</td>
</tr>
<tr>
<td>f or F</td>
<td>Real</td>
<td>Real, fixed-point format</td>
<td>191.04, -4.789</td>
</tr>
<tr>
<td>g or G</td>
<td>Real</td>
<td>Real, exponential, or decimal format, whichever format results in the shortest printed output</td>
<td>9.6001, 7.34E-8, -23.1E6</td>
</tr>
<tr>
<td>h or H</td>
<td>Integer</td>
<td>Hexadecimal format</td>
<td>3e, 262, a38, fff, 3E, A38</td>
</tr>
<tr>
<td>o or O</td>
<td>Integer</td>
<td>Octal format</td>
<td>127, 777</td>
</tr>
</tbody>
</table>
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 s gives %s", "s string");
        $strobe("Format S gives %S", "S string");
        $strobe("newline, \ntab, \tbackslash, \\
        ");
        $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
Format e gives 1.234568e+02
Format E gives 1.234568e+02
Format f gives 123.456789
Format F gives 123.456789
```

<table>
<thead>
<tr>
<th>format_character</th>
<th>Type of Argument</th>
<th>Output</th>
<th>Example Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>r</code> or <code>R</code></td>
<td>Real</td>
<td>Engineering notation format</td>
<td>123,457M, 12.345K</td>
</tr>
<tr>
<td><code>s</code> or <code>S</code></td>
<td>String constant</td>
<td>String format</td>
<td></td>
</tr>
</tbody>
</table>
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 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 166.

$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 166.

$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 166.

**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

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, $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 172. 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

```verilog
integer myChanDesc;
myChanDesc = $fopen ( "myfile" );
```

### Special $fopen Formatting Commands

The following special output formatting commands are available for use with the $fopen function.

<table>
<thead>
<tr>
<th>Command</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%C</td>
<td>Design filename</td>
<td>input.scs</td>
</tr>
<tr>
<td>%D</td>
<td>Date (yy-mm-dd)</td>
<td>94-02-28</td>
</tr>
<tr>
<td>%H</td>
<td>Host name</td>
<td>hal</td>
</tr>
<tr>
<td>%S</td>
<td>Simulator type</td>
<td>spectre</td>
</tr>
<tr>
<td>%P</td>
<td>Unix process ID #</td>
<td>3641</td>
</tr>
<tr>
<td>%T</td>
<td>Time (24hh:mm:ss)</td>
<td>15:19:25</td>
</tr>
<tr>
<td>%I</td>
<td>Instance name</td>
<td>opamp3</td>
</tr>
<tr>
<td>%A</td>
<td>Analysis name</td>
<td>dc0p,timeDomain,acSup</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Modifier</th>
<th>Extracted information</th>
</tr>
</thead>
<tbody>
<tr>
<td>:r</td>
<td>Root (base name) of the path for the file</td>
</tr>
<tr>
<td>:e</td>
<td>Extension of the path for the file</td>
</tr>
<tr>
<td>:h</td>
<td>Head of the path for any portion of the file before the last /</td>
</tr>
<tr>
<td>:t</td>
<td>Tail of the path for any portion of the file after the last /</td>
</tr>
<tr>
<td>::</td>
<td>The (:) character itself</td>
</tr>
</tbody>
</table>

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
```

by using different formatting commands and modifiers.

<table>
<thead>
<tr>
<th>Command and Modifiers</th>
<th>Resulting Opened File</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$fopen(&quot;%C&quot;)</code></td>
<td>None, because the design file cannot be overwritten.</td>
</tr>
<tr>
<td><code>$fopen(&quot;%C:r&quot;)</code></td>
<td>`/users/maxwell/circuits/opamp</td>
</tr>
<tr>
<td><code>$fopen(&quot;%C:e&quot;)</code></td>
<td><code>ckt</code></td>
</tr>
<tr>
<td><code>$fopen(&quot;%C:h&quot;)</code></td>
<td>`/users/maxwell/circuits</td>
</tr>
<tr>
<td><code>$fopen(&quot;%C:t&quot;)</code></td>
<td><code>opamp.ckt</code></td>
</tr>
<tr>
<td><code>$fopen(&quot;%C::&quot;)</code></td>
<td><code>/users/maxwell/circuits/opamp.ckt:</code></td>
</tr>
<tr>
<td><code>$fopen(&quot;%C:h:h&quot;)</code></td>
<td><code>/users/maxwell</code></td>
</tr>
<tr>
<td><code>$fopen(&quot;%C:t:r&quot;)</code></td>
<td><code>opamp</code></td>
</tr>
<tr>
<td><code>$fopen(&quot;%C:r:t&quot;)</code></td>
<td><code>opamp</code></td>
</tr>
<tr>
<td><code>$fopen(&quot;/tmp/%C:t:r.raw&quot;)</code></td>
<td><code>/tmp/opamp.raw</code></td>
</tr>
</tbody>
</table>
Reading from a File

Use the $fscanf function to read information from a file.

\[
fscanf\_function \::= \\
\quad fscanf ( \text{multi\_channel\_descriptor}, \text{"format"} \{ , storage\_arg \} )
\]

The \textit{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 \textit{storage\_arg}. The first \textit{storage\_arg} corresponds to the first nonskipping formatting command; the second \textit{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 166 for guidance.

For example, the following statement reads data from the file designated by \texttt{fptr1} and places the information in variables called \texttt{dbl} and \texttt{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.
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 166 for guidance.

For example, the following code fragment illustrates how you might write simultaneously to two open files.

```verilog
global
integer mcd1 ;
global
integer mcd2 ;
global
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") ;
```

### $fdisplay

Use the `$fdisplay` function to write information to a file.

```verilog
fdisplay_function ::= $fdisplay (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 166 for guidance.

### $fwrite

Use the `$fwrite` function to write information to a file.

```verilog
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 166 for guidance.

The `$fwrite` function does not insert automatic carriage returns in the output.

### 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.

### Exiting to the Operating System

Use the `$finish` function to make the simulator exit and return control to the operating system.

```
finish_function ::= $finish [ ( msg_level ) ] ;
```

```
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.

<table>
<thead>
<tr>
<th><code>msg_level</code></th>
<th>Messages printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Simulation time and location</td>
</tr>
<tr>
<td>2</td>
<td>Simulation time, location, and statistics about the memory and CPU time used in the simulation</td>
</tr>
</tbody>
</table>

**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 ;
```
Entering Interactive Tcl Mode

Use the $stop function to make the simulator enter interactive mode and display a Tcl prompt.

\[
\text{stop\_function ::= } \$stop \[(\text{msg\_level})]\; \\
\text{msg\_level ::= } \begin{array}{c}
0 \\
1 \\
2
\end{array}
\]

The msg_level value determines which diagnostic messages print before the simulator starts the interactive mode. The default msg_level value is 1.

<table>
<thead>
<tr>
<th>msg_level</th>
<th>Messages printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Simulation time and location</td>
</tr>
<tr>
<td>2</td>
<td>Simulation time, location, and statistics about the memory and CPU time used in the simulation</td>
</tr>
</tbody>
</table>

For example, to make the simulator go interactive, you might code

\>$stop\;$

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.

Declaring an Analog User-Defined Function

To define an analog function, use this syntax:

\[
\text{analog\_function\_declaration ::= } \\
\text{analog function [ type ] function\_identifier ;} \\
\text{function\_item\_declaration {function\_item\_declaration}} \\
\text{statement} \\
\text{endfunction} \\
\text{type ::= } \begin{array}{c}
\text{integer} \\
\text{real} \\
\end{array} \\
\text{function\_item\_declaration ::= } \\
\text{input\_declaration} \\
\text{block\_item\_declaration}
\]
block_item_declaration ::=  
  integer_declaration  
  |  real_declaration

**type** is the type of the value returned by the function. The default value is **real**.

**statement** cannot include analog operators and cannot define module behavior. Specifically, **statement** cannot include

- ddt operator
- idt operator
- idtmod operator
- Access functions
- Contribution statements
- Event control statements
- Simulator library functions, except that you can include the functions in the next list

**statement** can include references to

- $vt
- $vt(temp)
- $temperature
- $realtime
- $abstime
- analysis
- $strobe
- $display
- $write
- $fopen
- $fstrobe
- $fdisplay
- $fwrite
- $fclose
All mathematical functions

You can declare local variables to be used in the function.

Each function you define must have at least one declared input. Each function must also assign a value to the implicitly defined internal variable with the same name as the function.

For example,

```verbatim
analog function real chopper;
    input sw, in ; // The function has two declared inputs.
    real sw, in ;
    // The next line assigns a value to the implicit variable, chopper.
    chopper = ((sw > 0) ? in : -in);
endfunction
```

The `chopper` function takes two variables, `sw` and `in`, and returns a real result. You can use the function in any subsequent function definition or in the module definition.

Calling a User-Defined Analog Function

To call a user-defined analog function, use the following syntax.

```verbatim
analog_function_call ::= function_identifier ( expression , expression )
```

`function_identifier` must be the name of a defined function. Each `expression` is evaluated by the simulator before the function runs. However, do not rely on having expressions evaluated in a certain order because the simulator is allowed to evaluate them in any order.

An analog function must not call itself, either directly or indirectly, because recursive functions are illegal. Analog function calls are allowed only inside of analog blocks.

The module `phase_detector` illustrates how the `chopper` function can be called.

```verbatim
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;
        chopper = ((sw > 0) ? in : -in);
    endfunction
analog
    V(if0) <+ gain * chopper(V(lo),V(rf)); // Call from within the analog block.
endmodule
```

```
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. Module declarations cannot nest 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 G, “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 234.

The following sections discuss

- Instantiating Verilog-A Modules on page 182
- Connecting the Ports of Module Instances on page 184
- Overriding Parameter Values in Instances on page 185
- Instantiating Analog Primitives on page 188
- Using Inherited Ports on page 189
- Using an m-factor (Multiplicity Factor) on page 190
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 }
ordered_port_connection ::= [ 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 185.

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
```

Two of these gain blocks are connected, with the output of the first becoming the input of the second. The schematic looks like this.

```
qin ------- vd1 -------- aal ------- vd2 ------- qout
```

[Diagram of the circuit with two gain blocks connected]
This higher-level component is described by module \texttt{vquad}, which creates two instances, named \texttt{vd1} and \texttt{vd2}, of module \texttt{vdoubler}. Module \texttt{vquad} also defines external ports corresponding to those shown in the schematic.

\begin{verbatim}
module vquad (qin, qout);
    input qin;
    output qout;
    electrical qin, qout;
    wire aal;
    v doubler vd1 (qin, aal);
    v doubler vd2 (aal, qout);
endmodule
\end{verbatim}

### 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 \texttt{vquad}, in the previous example, uses an ordered list, where instance \texttt{vd1}'s first actual port name \texttt{qin} maps to \texttt{vdoubler}'s first formal port name \texttt{in}. Instance \texttt{vd1}'s second actual port name \texttt{aal} maps to \texttt{vdoubler}'s second formal port name, and so on.

### Mapping Ports with Ordered Lists

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 \texttt{child} and the module \texttt{instantiator} that instantiates it.

\begin{verbatim}
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;
    child child1 (conin [1:4], conin [6], conout);
end module
\end{verbatim}

You can tell from the order of port names in these modules that port \texttt{ina[0]} in module \texttt{child} maps to port \texttt{conin[1]} in instance \texttt{child1}. Similarly, port \texttt{inb} in \texttt{child} maps to port \texttt{conin[6]} in instance \texttt{child1}. Port \texttt{out} in \texttt{child} maps to port \texttt{conout} in instance \texttt{child1}.
Connecting the Ports of Module Instances

Developing modules that describe components is an important step on the way to the overall goal of simulating a system. But an equally important step is combining those components together 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 ;
    electrical qin, qout ;
    wire aal ;
    vdoubler vd1 (qin, aal) ;
    vdoubler vd2 (aal, qout) ;
endmodule
```

This time, note how the module instantiation statements in `vquad` use port names to establish a connection between output port `aal` of instance `vd1` and input port `aal` of instance `vd2`.

Module instantiation 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 67.

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 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 instantiation statement is

\[
\text{module_or_paramset_id} \ [ \ \text{parameter_value_assignment} \ ] \ \text{instance_list}
\]

The following sections discuss the \text{parameter_value_assignment} expression, which is further defined as

\[
\text{parameter_value_assignment} ::= \\
\quad | \ ( \ \text{named_param_override_list} ) \\
\text{named_param_override_list} ::= \\
\quad \text{named_param_override} \ ( , \ \text{named_param_override} ) \\
\text{named_param_override} ::= \\
\quad . \ \text{parameter_identifier} \ ( \ \text{expression} )
\]

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 instantiation statement itself. You can also use paramsets, as described in “Overriding Parameter Values by Using Paramsets” on page 186.

Overriding Parameter Values from the Instantiation Statement

Using the module instantiation statement, you can assign values to parameters by explicitly referring to parameter names. The new values must be constant expressions.
Overriding Parameter Values By Name

You override parameter values in an instantiated module by pairing the parameter names to be changed with the values they are to receive. A period and the parameter name come first in each pair, followed by the new value in parentheses. The parameter name must be the name of a parameter in the defining module of the module being instantiated. When you override parameter values by name, you are not required to specify values for every parameter.

Consider this modified definition of module \texttt{vdoubler}. This version has three parameters, \texttt{parm1}, \texttt{parm2}, and \texttt{parm3}.

\begin{verbatim}
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

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
\end{verbatim}

The module instantiation statement for instance \texttt{vd1} overrides parameter \texttt{parm3} by name to specify that the value for \texttt{parm3} should be changed to 4.0. The other two parameters retain the default values 0.2 and 0.1.

Overriding Parameter Values by Using Paramsets

The syntax for creating paramsets is given in “Paramsets” on page 60. This section discusses how to use paramsets to override parameter values.

The syntax for module instantiation is

\begin{verbatim}
module_instantiation ::= 
  module_or_paramset_id [ parameter_value_assignment ] instance_list
\end{verbatim}

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.

\begin{verbatim}
module baseModule (in out);
inout in, out;
electrical in, out;
\end{verbatim}
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:

- 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.

```verilog
module ri_test (pwr, gnd) ;
external pwr, gnd ;
parameter real ibias = 10u, ampl = 1.0 ;
external 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 instantiation 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:

```verilog
module fm_demodulator(vin, vout, vgnd) ;
input vin, vgnd ;
output vout ;
external vin, vout, vgnd ;
parameter real gain = 1 ;
  svcvs #(gain(gain),.poles({-1M, 0, -1M, 0}))
    af_filter (vout, vgnd, vin, vgnd) ;
```
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.

```verilog
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.

```verilog
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.

```verilog
// VerilogA for amslib, inv, veriloga
'include "constants.h"
'include "discipline.h"
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

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.

For more information, see the Inherited Connections Flow Guide.

Using an m-factor (Multiplicity Factor)

An m-factor is a value that can be inherited down a hierarchy of instances. Circuit designers use m-factors to mimic parallel copies of identical devices without having to instantiate large sets of devices in parallel. 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. If there are no passed m-factors in the instance or in the ancestors of the instance, the value of the m-factor is one.
To enable m-factors in Verilog-A, the simulator supports a Cadence attribute called `inherited_mfactor`, which is used to access the value of the m-factor.

**Accessing an Inherited m-factor**

To use an inherited m-factor, you use the `inherited_mfactor` attribute on a parameter declaration. Using this attribute on a parameter declaration sets the value of the parameter to the value of the m-factor inherited by the module.

For example, the following statement illustrates how to access an m-factor parameter called `m`.

```verilog
(* inherited_mfactor *) parameter real m=1;
```

**Example: Using an m-factor**

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.

The following module is defined in a file called `mfactor_res.va`.

```verilog
// 'include "discipline.h"
'include "constants.h"
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
```
Module `mf_res` is used in the following netlist.

```verbatim
//
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
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 you simulate this netlist, it generates results like the following, reflecting the division by `m` that appears in the `mf_res` module.

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
Controlling the Compiler

This chapter describes how to use the Cadence® Verilog®-A compiler directives for a range of tasks, including

- Implementing Text Macros on page 194
- Compiling Code Conditionally on page 196
- Including Files at Compilation Time on page 196
- Setting Default Rise and Fall Times on page 197
- Resetting Directives to Default Values on page 197

This chapter also describes predefined macros that you can use to determine the support provided by your simulator.

- Checking the Simulator Version on page 198
- Checking Support for Compact Modeling Extensions on page 198
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

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`.

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.

*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 compiler directive to control the inclusion or exclusion of code at compilation time.

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.

You can use an `ifdef compiler directive anywhere in your source description. You can, in fact, nest an `ifdef directive inside another `ifdef 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_compilerDirective ::= `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.

### Checking the Simulator Version

Use the `CDS_MMSIM6_0_OR_LATER` macro to check whether the simulator you are using is version 6.0 or later.

```markdown
CDS_MMSIM6_0_OR_LATER macro call::=
  `ifdef CDS_MMSIM6_0_OR_LATER

The `CDS_MMSIM6.0_OR_LATER` macro is predefined in the `disciplines.vams` file, so all you need to do is reference the macro. You can use this macro to choose different Verilog-A statements to be used in a module when the simulator version is 6.0 or greater.

### Checking Support for Compact Modeling Extensions

Use the `__VAMS_COMPACT_MODELING__` macro to determine whether the simulator supports the compact modeling extensions. The AMS simulator supports these extensions and sets the value of this macro to `t`.

```markdown
VAMS_COMPACT_MODELING macro call::=
  `ifdef lorenplorenlp

The `__VAMS_COMPACT_MODELING__` macro is predefined, so all you need to do is call the macro. (Notice the double underscore characters at both the beginning and the end of the macro name.) The returned value is `t` if the simulator supports the compact modeling extensions, which are:

- Output variables
- Attributes for parameter descriptions and units (`desc, units`)
- Net descriptions
- Modules (module description attribute)
- String parameters
- Parameter aliases
- Environment parameter functions (`$simparam`)
- Derivative operator (ddx)
- Limiting function ($limit)
- Hierarchy detections functions ($param_given)
- Display tasks ($debug)
- Format specifications (%r, %R)
- Local parameters (localparam)

If the simulator does not support the compact modeling extensions, the returned value is nil.
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 202
- Using Escaped Names in the Cadence Analog Design Environment on page 214
- Defining Quantities on page 214
- Using Multiple Cellviews for Instances on page 216
- Multilevel Hierarchical Designs on page 234
- Using Models with Verilog-A on page 239
- Saving Verilog-A Variables on page 240
- Displaying the Waveforms of Variables on page 240

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 208.

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.

- Type the name of the library.
- Type the name of the directory where you want the library saved.
- Unless you use these components for layout, you do not need a technology file.
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.


The New Library form opens. This form is different from the New Library form that you can open from the CIW.

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 tool 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 205.

However you create the symbol, it must reside in an existing library as described in “Preparing a Library” on page 202.
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 70.

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.

<table>
<thead>
<tr>
<th>Add Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hide</td>
</tr>
<tr>
<td>Library</td>
</tr>
<tr>
<td>Names</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>If Block Shape is set to freeform</th>
<th>If Block Shape is set to anything else</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>Drag the predefined block to the location where you want to place it and click. Refer to the Virtuoso Schematic Editor User Guide for details about modifying the block samples using the schBlockTemplate variable in the schConfig.il file.</td>
</tr>
</tbody>
</table>

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:
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.

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.

```verilog
//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:

```verbatim
//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
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.
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 form](image)

   - **Library Name**: ahdll
   - **Cell Name**: vdbase
   - **View Name**: verilogase
   - **Tool**: VerilogA-Editor
   - **Library path file**: /usr1/barbaral/cds.lib

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.

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.

<table>
<thead>
<tr>
<th>Symbol Generation Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
</tr>
<tr>
<td>Library Name</td>
</tr>
<tr>
<td>Interface</td>
</tr>
<tr>
<td>Pin Specifications</td>
</tr>
<tr>
<td>Left Pins</td>
</tr>
<tr>
<td>Right Pins</td>
</tr>
<tr>
<td>Top Pins</td>
</tr>
<tr>
<td>Bottom Pins</td>
</tr>
</tbody>
</table>

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 211.

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.

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 47, 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).

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**

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Required or Optional?</th>
<th>The value must be</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstol</td>
<td>Required</td>
<td>A real value</td>
</tr>
<tr>
<td>blowup</td>
<td>Optional</td>
<td>A real value</td>
</tr>
<tr>
<td>description</td>
<td>Optional</td>
<td>A string</td>
</tr>
<tr>
<td>huge</td>
<td>Optional</td>
<td>A real value</td>
</tr>
<tr>
<td>name</td>
<td>Required</td>
<td>A string</td>
</tr>
<tr>
<td>units</td>
<td>Optional</td>
<td>A string</td>
</tr>
</tbody>
</table>

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="Ω" 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.
Using Multiple Cellviews for Instances

As you develop a design, you might find it useful to have more than one verilog-a cellview for a given instance of a component. For example, you might want to have two or more verilog-a 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 verilog definitions of the cell ahdlTest. One of the definitions, which is assumed to have the view name verilogaone, is defined by the module

```verilog
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 tool 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.

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 tool 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.

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*. 

---

December 2006 219 Product Version 6.1
The Edit Object Properties form opens.

```
| Apply To | only current | instance |
| Show     | system       | user     | CDF       |
```

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.


   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:

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.

   ![Select View Menu](image)

   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.

   ![Schematic Diagram]

   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.

   ![Descend Dialog]

   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.

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.
As a check, you can use the capabilities of the analog design environment Simulation window to generate a netlist.

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.

```
// Generated for: spectre
// 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
I1 (net5 net10) gainvera
R0 (net5 0) resistor r=1K
R1 (net10 0) resistor r=1K
simulatorOptions options rettol=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"
```

"include "constants.vams"
"include "disciplines.vams"
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 $I_1$ 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.
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.

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.
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.

```
// Generated for: spectre
// Generated on: Sep 14 13:29:12 1996
// 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/artise/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 scales=1.0 scale=1.0 gmin=1e-12 rforce=1 maxnotes=5 \nmaxwarns=5 digits=5 cols=80 pivrel=1e-3 ccktctool=1800 \nsensfile="..//psf/sens.output"modelParameter info what=holds where=rawfile
element info what=inst where=rawfile
outputParameter info what=output where=rawfile
saveOptions options save=allpub
ahdl_include "/old2/lorenp/demogain/ahdl/qahdl.def"
ahdl_include "/old2/lorenp/demogain/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*. 
The instances within `demogain` appear in the `Instance Bindings` table.

<table>
<thead>
<tr>
<th>Inst Name</th>
<th>Library</th>
<th>Cell</th>
<th>View Found</th>
<th>View To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>I0</td>
<td>AHDL</td>
<td>gain</td>
<td>ahd1</td>
<td>ahd1</td>
</tr>
<tr>
<td>I1</td>
<td>AHDL</td>
<td>gain</td>
<td>veriloga</td>
<td></td>
</tr>
<tr>
<td>R0</td>
<td>analogLib</td>
<td>res</td>
<td>spectre</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td>analogLib</td>
<td>res</td>
<td>spectre</td>
<td></td>
</tr>
<tr>
<td>I2</td>
<td>basic</td>
<td>gnd</td>
<td>schematic</td>
<td></td>
</tr>
</tbody>
</table>

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 – ahd1`.
   
   The `View Found` and the `View To Use` fields both change to `ahd1`.

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 220, 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 ahdI, not veriloga.

3. Switch the CDF Parameter of view field to ahdI.

   The parameter values set for instance I0 while it was bound with the ahdI cellview appear. If you rebind the ahdI cellview with instance I0, the ahdI 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 ahdI cellviews. When gain in the ahdI 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 $I_0$ instance is bound with the veriloga cellview. The netlist also shows that the $gain$ parameter has the expected value of 5.

```
// Generated for: spectre
// Design library name: AHDL
// Design cell name: demogain
// Design view name: config
simulator lang=spec
global
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
$I_0 (net10 0) gainvera gain=5$
$I_1 (net10 net5) gainahdl$
$R_0 (net5 0) resistor r=1K$
$R_1 (net10 0) resistor r=1K$
simulatorOptions options reltol=1e-3 abstol=1e-6 iabstol=1e-12 temp=27 \ 
tnom=27 scalem=1.0 scale=1.0 gm=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 netlist 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)
node[V,I] R1, ref, out, CA, CB, VCC, vControl;
{
    node [V, I] cntrl;
    real state;
    VCOshape shape (ref, cntrl, VCC, vControl);
    resistor RX (CB, ref) (r=.001);
    resistor R1min (cntrl, R1) (r=500);
    capacitor Cmin (CA, CB) (c=10p);
```

initial {
    state = 1.0;
}

analog {
    if ($analysis("dc") || $time() == 0.0) {
        val(CA, CB) <- 0.0;
    }
    if ($threshold(val(CA)+1.0, -1)) {
        state = 1.0;
    }
    if ($threshold(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);
}

The VCO2 module is part of a larger schematic, which produces the following netlist:

**Instantiation of VCO2 in the top-level design**

```verbatim
// 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 scalel=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" opoint=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/ahdldemo/QPSK/VCO2/ahdl/ahdl.def"
ahdl_include "/net/cds9886/u1/public/ahdldemo/QPSK/VCOshape/ahdl/ahdl.def"
ahdl_include "/net/cds9886/u1/public/ahdldemo/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
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.

```verilog
module filter (in, out);
  extrn inst1 (pinone, poutone);
  analog begin
    ... 
  end
endmodule
```

Instantiate the schematic like a child module, but with no parameters.

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 cmos_sch_cmos_sch_schematic_veriloga_ahdl

Stop View List
spectre
```
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.

```vhd
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:

1. In the simulation control window, choose *Outputs – Save All*. The *Outputs – Save All* command opens the Save Options form. In that form, click the *all* button 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
   ```
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:

\[
\text{hier\_name ::= } \hspace{1cm} [ \text{instance\_name} \{ . \text{instance\_name} \} ] \text{HDL\_Instance\_name}
\]

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 \hspace{0.5cm} \rightarrow \hspace{0.5cm} i7.i2.i3
Verilog-A instance below one block \hspace{0.5cm} \rightarrow \hspace{0.5cm} i2.i3

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

\[
\text{save instance\_name.internal\_node\_name}
\]

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

   \[
   \text{simulation/design\_name/spectre/view\_name}
   \]

   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.
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.
Advanced Modeling Examples

This chapter examines in detail several examples that use the Cadence® Verilog®-A language to model complex systems. Two electrical modeling examples are presented first. For an example using Verilog-A to model a mechanical system, see “Mechanical Modeling” on page 255.

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**

![Rectifier Circuit Diagram]

**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.

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.

```verilog-a
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 ) )

    // 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
```

December 2006 245 Product Version 6.1
The transformers are modeled with the following module, which includes leakage inductance effects:

```verilog
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.

```verilog
`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);
  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.

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.

```
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
```

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,
so both the front and back gate potentials influence channel conductivity. This implementation does not model short-channel effects.

The module definition is

```verilog
#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 tox = 0.5u;
parameter real nsub = 1e14;
parameter real ngate = 1e19;
parameter real nbody = 5e15;
```
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(n gate/ni) - phi/2;
    wkb = $vt*ln(nsub/ni) - phi/2;
  end
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);
alp = cbb/cof;

vtfa = vfbf + (1 + cb/cof)*phi - qb/(2*cof);
vgba = dev_type_sign*vfbb - phi*cbo/cbb - qb/(2*cbo);
vghi = dev_type_sign*vfbb + phi - qb/(2*cbo);

kp = width*u0*1e-4*cof/length;
back_surf = 0;
end     // of initial_step code

// the following code is executed at every iteration

vgs = dev_type_sign*V(vgate_front, vsource);
vds = dev_type_sign*V(vdrain, vsource);
vgsb = dev_type_sign*V(vgate_back, vsource);

// calc. threshold and saturation voltages.
//
vtff = vtfa - (vgsb - vgbi)*cbb/cof;
vdsat = (vgs - vtff)/(1 + alp);

//
// drain current calculations.
//
if (vgs < vtff) begin

    // front-channel in accumulation / cutoff region(s).
    //
    id = 0;
    qn = 0;
    qd = 0;
    qs = 0;
    qgf = width*length*cfbb*(vgs - wkf - qb/(2*cbb)
               - (vgsb - vfbb + qb/(2*cob)))
    qgb = - (qgf + width*length*qb);

end else if (vds < vdsat) begin

    // front-channel in linear region.
    //
    id = kp*((vgs - vtff)*vds - 0.5*
             (1 + cbb/cof)*vds*vds);
    id = id*(1 + lambda*vds);
    tmp1 = (1 + alp)*vds;
    qn = -width*length*cof*(vgs - vtff - tmp1/2 +
                              tmp1*tmp1/ (12*(vgs - vtff - tmp1/2)));
    qd = 0.4*qn;
    qs = 0.6*qn;

end
```verilog
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.
    end else if (back_surf == 2) begin
        $display("Back-surface went into accumulation.
    end
end
endmodule
```
The netlist file instantiates an n-channel TFT device with a width of 2 microns (2µ) and a length of 1 micron (1µ). The drain-source voltage (vds) 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 vsourc dc=5
vbs back_gate source vsourc dc=-3
vgs gate source vsourc 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 255.

**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 243.

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.

```verbatim
// spring.va
`include "disciplines.vams"
`include "constants.vams"
```
module spring (posp, posn);
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
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.

```verilog
// 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 = $realtime - 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;
    end else if (Time < duration) begin
      Pos(posin) <+ height/6  * (1 - Time / duration);
    end else begin
      Pos(posin) <+ 0;
    end
  end
endmodule
```

Limiter

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.
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 \( \text{maxl} \) and greater than \( \text{minl} \), the model must be in the normal operating region. If the length is less than or equal to \( \text{minl} \), the limiter has bottomed out. However, because of the limiting, the length cannot be less than \( \text{minl} \), so the limiter bottoms out if the length equals \( \text{minl} \). 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.

```verbatim
// 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
```

```
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", $realtime );
  end
end else begin
  if (too_long) begin
    $strobe( "%M: Topped out at t = %E s.\n", $realtime );
    out_of_range = 1;
  end
  else if (too_short) begin
    $strobe( "%M: Bottomed out at t = %E s.\n", $realtime );
    out_of_range = 1;
  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® 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);
```

![Force vs Position Graph](image-url)
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", $realtime );
        end
    end else begin
        if (flying) begin
            $strobe( "\%M: Airborne at t = \%E s.\n", $realtime );
            reported = 1;
        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.
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.

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.

```verbatim
// 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
```

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

\[ \tau_{ext} + rF_{teeth} + I\alpha = 0 \]
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.

![Diagram of gears](image)

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

\[
2\pi r_1 \theta_1 = -2\pi r_2 \theta_2
\]

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

\[ \frac{\tau_1}{r_1} = \frac{\tau_2}{r_2} \]

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.

\[ \tau_2 = \frac{r_2}{r_1} \tau_1 \]

Finally, the effect of the inertia of the gear and shaft is added.

\[ \tau = \dot{\tau} + I\alpha \]

where \( \tau \) is the total torque applied externally to the shaft, \( \dot{\tau} \) 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:

\[ \tau_2 = I_2 \alpha_2 + \frac{r_2}{r_1}(\tau_1 - I_1 \alpha_1) \]

**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 inertial1=0 from [0:inf);
parameter real radius2=1 from (0:inf);
parameter real inertial2=0 from [0:inf);

    analog begin
```

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, s1 and s2, represent shafts.

```verilog
// 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

![Graph showing the transient response of the gearbox with two curves, s1 and s2, against time. The graph has a y-axis labeled 'Rotational Velocity' ranging from 0.0 to 0.8 and an x-axis labeled 'Time' ranging from 0.0 to 2.0. The curve s1 starts at 0.0 and increases linearly to 0.8, while s2 starts at 0.0 and increases more gradually to 0.8.]
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 270
- Simulating a System on page 271
Kirchhoff’s Laws

Simulation of Verilog®-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**

\[ \text{Kirchhoff’s Flow Law} \]

\[ \text{flow}_1 + \text{flow}_2 + \text{flow}_3 = 0 \]

\[ \text{Kirchhoff’s Potential Law} \]

\[ \text{potential}_1 + \text{potential}_2 + \text{potential}_3 + \text{potential}_4 = 0 \]
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 \( (dq/dt) \) with a discrete-time finite difference approximation. The simulation time interval is discretized and solved at individual time points along the interval. The simulator controls the interval between the time points to ensure the accuracy of the finite difference approximation. At each time point, 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 29.) On the first iteration, the signal values used in Verilog-A expressions are approximate and do not satisfy Kirchhoff’s laws.

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, \( v^{(j)}(t) \), must be close to the proposed solution on the previous iteration, \( v^{(j-1)}(t) \), and

\[
|v_n^{(j)} - v_n^{(j-1)}| < \text{reltol} \max\left(\left|\frac{v_n^{(j)}}{v_n^{(j-1)}}\right|, \left|\frac{v_n^{(j-1)}}{v_n^{(j)}}\right|\right) + \text{abstol}
\]

where \( \text{reltol} \) is the relative tolerance and \( \text{abstol} \) is the absolute tolerance.

\( \text{reltol} \) 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 \( \text{abstol} \) attribute for the natures you use. The absolute tolerance is important when \( v_n \) is converging to zero. Without \( \text{abstol} \), the iteration never converges.

The second criterion ensures that Kirchhoff’s Flow Law is satisfied:

\[
\left|\sum_n f_n(v^{(j)})\right| < \text{reltol} \max\left(\left|f_n^i(v^{(j)})\right|\right) + \text{abstol}
\]

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 274
- Sources on page 275

For examples, see

- Linear Conductor on page 280
- Linear Resistor on page 281
- RLC Circuit on page 281
- Simple Implicit Diode on page 281
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

```
+ p |
```

The branch flow of a potential probe is zero.

The equivalent circuit model for a flow probe is

```
| f |
```

The branch potential of a flow probe is zero.

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.
Port Branches

You can declare a port branch by specifying a port node twice in a branch declaration. For example, module portex declares a port branch called portbranch.

```verilog
module portex (inport, outport);
electrical inport, outport;
branch (outport, outport) portbranch; // Declares port branch
endmodule
```

The difference between a port branch and a simple port can be illustrated schematically as follows:

**Simple port**

```
Module
```

**Port branch**

```
Module
```

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 anode port branch becomes too large.

```verilog
module diode (a, c);
electrical a, c;
branch (a, c) diode, cap;
branch (a, a) anode; // Declares a port branch
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(anode) > 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 277.

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

```verilog
branch (ps,ns) in, (p,n) out;
V(out) <+ A * I(in);
```

The model for a *current-controlled current source* is

```verilog
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.

```verilog
if (closed)
  V(p,n) <+ 0 ;
else
  I(p,n) <+ 0 ;
```

Alternatively, you could achieve the same result with

```verilog
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`.

```verilog
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 119.

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.

```verilog
analog begin
    V(out) <+ 1.0; // Discarded
    I(out) <+ 1.0; // Discarded
    V(out) <+ 1.0;
end
```

In the next example,
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.h"
#include "discipline.h"
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
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 \( I\text{(cond)} \) makes \( \text{cond} \) a current (flow) source branch, and \( V\text{(cond)} \) accesses the potential probe built into the current source branch.
### Linear Resistor

The model for a linear resistor is

```verilog
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 \( V(\text{res}) \) makes \( \text{res} \) a potential source branch. \( I(\text{res}) \) 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.

\[
v(t) = R \dot{i}(t) + L \frac{d}{dt}i(t) + \frac{1}{C}\int_{-\infty}^{t} i(\tau)d\tau
\]

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.

\[
I(a,c) <+ \text{is} * (\text{limexp}((V(a,c) - rs * I(a,c)) / \text{vt}) - 1) ;
\]
Standard Definitions

The following definitions are included in the disciplines.vams and constants.vams files, which are supplied with the Cadence® Verilog®-A language. To see the contents of these files, go to

- disciplines.vams File on page 284
- constants.vams File on page 288

You can use these definitions as they are, change them, or override them. For example, to override the default value of the abstol attribute of the nature current, define CURRENT_ABSTOL before including the disciplines.vams file.

For information on how to include these definitions in your files, see “Including Files at Compilation Time” on page 196.
disciplines.vams File

`ifdef DISCIPLINES_VAMS
`else
`define DISCIPLINES_VAMS 1

// Release/Version related macros
`ifdef CDS_MMSIM6_0_KERNEL_OR_LATER
`else
`define CDS_MMSIM6_0_KERNEL_OR_LATER 1
`endif

/*
* Verilog-AMS HDL version 2.2 introduced a number of extensions
* to support compact modeling.
* Predefined macros here.
*/
`ifdef __VAMS_COMPACT_MODELING__
`else
`define __VAMS_COMPACT_MODELING__ 1
`endif

// Natures and Disciplines

`ifdef VAMS_ELEC_DIS_ONLY
`else
discipline \logic
domain discrete;
enddiscipline
`endif

/*
* Default absolute tolerances may be overridden by setting the
* appropriate _ABSTOL prior to including this file
*/

// Electrical

// Current in amperes
nature Current
  units = "A";
  access = I;
`ifdef VAMS_ELEC_DIS_ONLY
`else
  idt_nature = Charge;
`endif
`ifdef CURRENT_ABSTOL
  abstol = `CURRENT_ABSTOL;
`else
  abstol = 1e-12;
`endif
endnature

`ifdef VAMS_ELEC_DIS_ONLY
`else
// Charge in coulombs
nature Charge
  units = "coul";
  access = Q;
  ddt_nature = Current;
`ifdef CHARGE_ABSTOL
  abstol = `CHARGE_ABSTOL;
`else
  abstol = 1e-12;
`endif
endnature
`endif
abstol = `CHARGE_ABSTOL;
`else
  abstol = 1e-14;
`endif
endnature
`endif

// Potential in volts
nature Voltage
  units = "V";
  access = V;
`ifdef VAMS_ELEC_DIS_ONLY
`else
  idt_nature = Flux;
`endif
`ifdef VOLTAGE_ABSTOL
  abstol = `VOLTAGE_ABSTOL;
`else
  abstol = 1e-6;
`endif
endnature
`ifdef VAMS_ELEC_DIS_ONLY
`else
// Flux in Webers
nature Flux
  units = "Wb";
  access = Phi;
  ddt_nature = Voltage;
`ifdef FLUX_ABSTOL
  abstol = `FLUX_ABSTOL;
`else
  abstol = 1e-9;
`endif
endnature
`endif

// Conservative discipline
discipline electrical
  potential Voltage;
  flow Current;
enddiscipline

// Signal flow disciplines
discipline voltage
  potential Voltage;
enddiscipline
discipline current
  potential Current;
enddiscipline

`ifdef VAMS_ELEC_DIS_ONLY
`else
// Magnetic
// Magnetomotive force in Ampere-Turns.
nature Magneto_Motive_Force
  units = "A*turn";
  access = MMF;
`ifdef MAGNETO_MOTIVE_FORCE_ABSTOL
  abstol = `MAGNETO_MOTIVE_FORCE_ABSTOL;
`else
  abstol = 1e-9;
`endif
endnature
`endif


// Conservative discipline

// Magnetic

discipline magnetic

potential Magneto_Motive_Force;
flow Flux;

enddiscipline

// Thermal

// Temperature in Celsius

nature Temperature

units = "C";
access = Temp;

`ifdef TEMPERATURE_ABSTOL
abstol = `TEMPERATURE_ABSTOL;
`else
abstol = 1e-4;
`endif

dannature

// Power in Watts

nature Power

units = "W";
access = Pwr;

`ifdef POWER_ABSTOL
abstol = `POWER_ABSTOL;
`else
abstol = 1e-9;
`endif

dannature

// Conservative discipline

discipline thermal

potential Temperature;
flow Power;

enddiscipline

// Kinematic

// Position in meters

nature Position

units = "m";
access = Pos;

ddt_nature = Velocity;

`ifdef POSITION_ABSTOL
abstol = `POSITION_ABSTOL;
`else
abstol = 1e-6;
`endif

dannature

// Velocity in meters per second

nature Velocity

units = "m/s";
access = Vel;

ddt_nature = Acceleration;

idt_nature = Position;

`ifdef VELOCITY_ABSTOL
abstol = `VELOCITY_ABSTOL;
`else


abstol     = 1e-6;
`endif
endnature

// Acceleration in meters per second squared
nature Acceleration
  units    = "m/s^2";
  access   = Acc;
  ddt_nature = Impulse;
  idt_nature = Velocity;
`ifdef ACCELERATION_ABSTOL
  abstol  = `ACCELERATION_ABSTOL;
`else
  abstol  = 1e-6;
`endif
endnature

// Impulse in meters per second cubed
nature Impulse
  units    = "m/s^3";
  access   = Imp;
  idt_nature = Acceleration;
`ifdef IMPULSE_ABSTOL
  abstol  = `IMPULSE_ABSTOL;
`else
  abstol  = 1e-6;
`endif
endnature

// Force in newtons
nature Force
  units    = "N";
  access   = F;
`ifdef FORCE_ABSTOL
  abstol  = `FORCE_ABSTOL;
`else
  abstol  = 1e-6;
`endif
endnature

// Conservative disciplines
discipline kinematic
  potential    Position;
  flow         Force;
enddiscipline

discipline kinematic_v
  potential    Velocity;
  flow         Force;
enddiscipline

// Rotational
// Angle in radians
nature Angle
  units     = "rads";
  access    = Theta;
  ddt_nature = Angular_Velocity;
`ifdef ANGLE_ABSTOL
  abstol   = `ANGLE_ABSTOL;
`else
  abstol   = 1e-6;
`endif
endnature
// Angular Velocity in radians per second
nature Angular_Velocity
  units = "rads/s";
  access = Omega;
  ddt_nature = Angular_Acceleration;
  idt_nature = Angle;
`ifdef ANGULAR_VELOCITY_ABSTOL
  abstol = `ANGULAR_VELOCITY_ABSTOL;
`else
  abstol = 1e-6;
`endif
endnature

// Angular acceleration in radians per second squared
nature Angular_Acceleration
  units = "rads/s^2";
  access = Alpha;
  idt_nature = Angular_Velocity;
`ifdef ANGULAR_ACCELERATION_ABSTOL
  abstol = `ANGULAR_ACCELERATION_ABSTOL;
`else
  abstol = 1e-6;
`endif
endnature

// Force in newtons
nature Angular_Force
  units = "N*m";
  access = Tau;
`ifdef ANGULAR_FORCE_ABSTOL
  abstol = `ANGULAR_FORCE_ABSTOL;
`else
  abstol = 1e-6;
`endif
endnature

// Conservative disciplines
discipline rotational
  potential    Angle;
  flow         Angular_Force;
enddiscipline

discipline rotational_omega
  potential    Angular_Velocity;
  flow         Angular_Force;
enddiscipline

`endif // VAMS_ELEC_DIS_ONLY
`endif // DISCIPLINES_VAMS
`ifdef VAMS_ELEC_DIS_ONLY
`undef VAMS_ELEC_DIS_ONLY
`undef DISCIPLINES_VAMS
`endif

constants.vams File

// Mathematical and physical constants
`ifdef CONSTANTS_VAMS
`else
  `define CONSTANTS_VAMS 1
`endif
// M_ is a mathematical constant
`define M_E 2.7182818284590452354
`define M_LOG2E 1.4426950408889634074
`define M_LOG10E 0.43429448190325182765
`define M_LN2 0.693147180555994530942
`define M_LN10 2.30258509299404568402
`define M_PI 3.14159265358979323846
`define M_TWO_PI 6.28318530717958647652
`define M_PI_2 1.57079632679489661923
`define M_PI_4 0.78539816339744830962
`define M_1_PI 0.31830988618379067154
`define M_2_PI 0.63661977236758134308
`define M_2_SQRTPI 1.12837916709551257390
`define M_SQRT2 1.41421356237309504880
`define M_SQRT1_2 0.70710678118654752440

// P_ is a physical constant
// charge of electron in coulombs
`define P_Q 1.6021918e-19
// speed of light in vacuum in meters/sec
`define P_C 2.997924562e8
// Boltzmann's constant in joules/kelvin
`define P_K 1.3806226e-23
// Planck's constant in joules*sec
`define P_H 6.6260755e-34
// permittivity of vacuum in farads/meter
`define P_EPS0 8.85418792394420013968e-12
// permeability of vacuum in henrys/meter
`define P_U0 (4.0e-7 * `M_PI)
// zero celsius in kelvin
`define P_CELSIUS0 273.15

`endif
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 293
- Basic Components on page 310
- Control Components on page 318
- Logic Components on page 326
- Electromagnetic Components on page 346
- Functional Blocks on page 349
- Magnetic Components on page 372
- Mathematical Components on page 376
- Measure Components on page 393
- Mechanical Systems on page 413
- Mixed-Signal Components on page 420
- Power Electronics Components on page 429
- Semiconductor Components on page 432
- Telecommunications Components on page 440

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.
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

\(i_{in\_p}, i_{in\_n}:\) differential input current terminals [V,A]

\(i_{out}:\) output current terminal [V,A]

Description

Outputs \(i_{leak}\) when differential input current \((i_{in\_p} - i_{in\_n})\) is between \(i_{dead\_low}\) and \(i_{dead\_high}\). When outside the deadband, the output current is an amplified version of the differential input current plus \(i_{leak}\).

Instance Parameters

\(i_{dead\_low} = \text{lower range of dead band [A]}\)

\(i_{dead\_high} = \text{upper range of dead band [A]}\)

\(i_{leak} = \text{offset current; only output in deadband [A]}\)

\(gain\_low = \text{differential current gain in lower region []}\)

\(gain\_high = \text{differential current gain in lower region []}\)
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 \( i_{clamp\_upper} \) and \( i_{clamp\_lower} \) of the input current.

Instance Parameters

\( i_{clamp\_upper} = \) upper clamping current [A]
\( i_{clamp\_lower} = \) 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

vin_p, vin_n: differential input voltage [V,A]

vout: output voltage [V,A]

vref: reference voltage [V,A]

vspply_p: positive supply voltage [V,A]

vspply_n: negative supply voltage [V,A]

Instance Parameters

gain = gain []

freq_unitygain = unity gain frequency [Hz]

rin = input resistance [Ohms]

vin_offset = input offset voltage referred to negative [V]

ibias = input current [A]

iin_max = maximum current [A]

rsrc = source resistance [Ohms]

rout = output resistance [Ohms]

vsoft = soft output limiting value [V]
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 faction 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-vgnd 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)]
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 \( c\_\text{mean} \) and a standard deviation of \( c\_\text{std} \).

Two seeds are needed to generate the gaussian distribution.

Instance Parameters

\( c\_\text{mean} = \) mean capacitance [Ohms]

\( c\_\text{dev} = \) standard deviation of capacitance [Ohms]

\( \text{seed1} = \) first seed value for randomly generating capacitance values []

\( \text{seed2} = \) second seed value for randomly generating capacitance values []

\( \text{show\_val} = \) 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 \( r_{\text{mean}} \) and a standard deviation of \( r_{\text{std}} \).

Two seeds are needed to generate the gaussian distribution.

Instance Parameters

\( r_{\text{mean}} = \) mean resistance [Ohms]

\( r_{\text{dev}} = \) standard deviation of resistance [Ohms]

\( \text{seed1} = \) first seed value for randomly generating resistance values []

\( \text{seed2} = \) second seed value for randomly generating resistance values []

\( \text{show\_val} = \) 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

\[ vout = \text{gain}_\text{const} \times (vctrl_p - vctrl_n) \times (vin_p - vin_n) + (vout_{\text{high}} + vout_{\text{low}})/2. \]

When there is an input offset voltage, \( \text{vin}_{\text{offset}} \) is subtracted from \( (vin_p - vin_n) \).

Instance Parameters

\( \text{gain}_\text{const} = \text{amplifier gain when } (vctrl_p - vctrl_n) = 1 \text{ volt} \]
\( \text{vout}_{\text{high}} = \text{upper output limit } [\text{V}] \]
\( \text{vout}_{\text{low}} = \text{lower output limit } [\text{V}] \]
\( \text{vin}_{\text{offset}} = \text{input offset } [\text{V}] \)
Basic Components

Resistor

Terminals

vp, vn: terminals (V,A)

Instance Parameters

r = resistance (Ohms)
Capacitor

Terminals
vp, vn: terminals (V,A)

Instance Parameters

\( c = \text{capacitance (F)} \)
Inductor

Terminals

vp, vn: terminals (V,A)

Instance Parameters

l = inductance (H)
Voltage-Controlled Voltage Source

Terminals

\texttt{vout\_p, vout\_n:} controlled voltage terminals [V,A]
\texttt{vin\_p, vin\_n:} controlling voltage terminals [V,A]

Instance Parameters

\texttt{gain = voltage gain []}
Current-Controlled Voltage Source

Terminals

\( v_{out\_p}, v_{out\_n} \): controlled voltage terminals [V,A]
\( i_{in\_p}, i_{in\_n} \): controlling current terminals [V,A]

Instance Parameters

\( r_m = \) 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

\[ \text{sigerr} = \text{sigset} - \text{sigact} \]

**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

tdel, trise, tfall = {usual}
Lag Compensator

Terminals

sigin: (val, flow)
sigout: (val, flow)

Description

\[ TF = \text{gain} \times \alpha \times \frac{1 + \tau \times S}{1 + \alpha \times \tau \times S} \]

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

\[ TF = \text{gain} \times \alpha \times \frac{1 + \text{tau} \times S}{1 + \alpha \times \text{tau} \times S} \]

**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/\text{tau}) [s]

alpha = compensator pole at -(1/(\alpha \times \text{tau})); \alpha < 1 []
Lead-Lag Compensator

Terminals

\[ \text{sigin: (val, flow)} \]

\[ \text{sigout: (val, flow)} \]

Description

\[ TF = \]
\[ \text{gain} \times \alpha_1 \times \frac{1 + \tau_1 \times s}{1 + \alpha_1 \times \tau_1 \times s} \times \alpha_2 \times \frac{1 + \tau_2 \times s}{1 + \alpha_2 \times \tau_2 \times s} \]

Defining larger values of \text{abstol} and \text{huge} for the quantities associated with \text{sigin} and \text{sigout} can help overcome convergence and clipping problems.

Instance Parameters

\[ \text{gain} = \text{compensator gain []} \]

\[ \text{tau1} = \text{compensator zero at } -(1/\text{tau1}) [s] \]

\[ \text{alpha1} = \text{compensator pole at } -(1/(\text{alpha} \times \text{tau1})); \text{alpha1} > 1 [] \]

\[ \text{tau2} = \text{compensator zero at } -(1/\text{tau2}) [s] \]

\[ \text{alpha2} = \text{compensator pole at } -(1/(\text{alpha} \times \text{tau2})); \text{alpha2} < 1 [] \]
Proportional Controller

Terminals

\[
sigin: \quad (\text{val, flow})
\]

\[
sigout: \quad (\text{val, flow})
\]

Description

\[
sigout = kp \times sigin
\]

**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

\[
kp = \text{proportional gain} \quad [\]
\]
Proportional Derivative Controller

Terminals

sigin:  (val, flow)
sigout:  (val, flow)

Description

\[ \text{sigout} = \text{kp} \times \text{sigin} + \text{kd} \times \dot{\text{sigin}} \]

**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 []
kd = differential gain []
Proportional Integral Controller

Terminals

\(\text{sigin: } (\text{val, flow})\)

\(\text{sigout: } (\text{val, flow})\)

Description

This model is a proportional, integral, and derivative controller.

\(\text{sigout} = \text{kp} \times \text{sigin} + \text{ki} \times \text{integ} (\text{sigin}) + \text{kd} \times \text{dot} (\text{sigin})\)

Note: Defining larger values of \text{abstol} and \text{huge} for the quantities associated with \text{sigin} and \text{sigout} can help overcome convergence and clipping problems.

Instance Parameters

\(\text{kp} = \text{proportional gain} []\)

\(\text{ki} = \text{integral gain} []\)
Proportional Integral Derivative Controller

Terminals

sigin:  (val, flow)
sigout:  (val, flow)

Description

\[ \text{sigout} = \text{kp} \times \text{sigin} + \text{ki} \times \text{integ}(\text{sigin}) + \text{kd} \times \text{dot}(\text{sigin}) \]

**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 []
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

\[
\begin{align*}
\text{vin:} & \quad [V,A] \\
\text{vout:} & \quad [V,A]
\end{align*}
\]

Instance Parameters

\[
\begin{align*}
\text{vlogic\_high} &= \text{output voltage for high [V]} \\
\text{vlogic\_low} &= \text{output voltage for high [V]} \\
\text{vtrans} &= \text{voltages above this at input are considered high [V]} \\
\text{tdel, trise, tfall} &= \{\text{usual}\} [s]
\end{align*}
\]
NOR Gate

Terminals

\begin{align*}
\text{vin1, vin2:} & \quad [V, A] \\
\text{vout:} & \quad [V, A]
\end{align*}

Instance Parameters

\begin{align*}
\text{vlogic\_high} & = \text{output voltage for high [V]} \\
\text{vlogic\_low} & = \text{output voltage for high [V]} \\
\text{vtrans} & = \text{voltages above this at input are considered high [V]} \\
\text{tdel, trise, tfall} & = \{\text{usual}\} [s]
\end{align*}
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]
tdel, trise, tfall = {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

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q</th>
<th>Q'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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]
JK-Type Flip-Flop

Terminals

vin_j, vin_k: inputs
vout_q, vout_qbar: outputs

Description

Triggered on the rising edge.

Logic Table

<table>
<thead>
<tr>
<th>J</th>
<th>K</th>
<th>Q</th>
<th>Q(t+e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

\( \text{sigout} = \text{sigin added to sigshift}. \)

Instance Parameters

\( \text{sigshift} = \text{level shift (val)} \)
RS-Type Flip-Flop

Terminals

\[
\begin{align*}
\text{vin}_s & : [V,A] \\
\text{vin}_r & : [V,A] \\
\text{vout}_q, \text{vout}_{qbar} & : [V,A]
\end{align*}
\]

Logic Table

<table>
<thead>
<tr>
<th>S(t)</th>
<th>R(t)</th>
<th>Q(t)</th>
<th>Q(t+e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>X</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>T</th>
<th>Q</th>
<th>Q(t+e)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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

\[ \text{vin1, vin2: inputs [V,A]} \]
\[ \text{vout\_diff: difference out [V,A]} \]
\[ \text{vout\_borrow: borrow out [V,A]} \]

Formula

\[ \text{vin1 - vin2 = vout\_diff and borrow} \]

Truth Table

<table>
<thead>
<tr>
<th>in1</th>
<th>in2</th>
<th>diff</th>
<th>borrow</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Instance Parameters

\[ \text{vlogic\_high = logic high value [V]} \]
\[ \text{vlogic\_low = logic low value [V]} \]
\[ \text{vtrans = threshold for inputs to be high [V]} \]
\[ \text{tdel, trise, tfall = \{usual\} [s]} \]
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

<table>
<thead>
<tr>
<th>in1</th>
<th>in2</th>
<th>bin</th>
<th>bout</th>
<th>doff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

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

\text{vp: positive terminal [V,A]}
\text{vn: negative terminal [V,A]}
\text{pos\_shaft: motor shaft [rad, Nm]}

Description

This is a model of a DC motor driving a shaft.

Instance Parameters

\text{km = motor constant [Vs/rad]}
\text{kf = flux constant [Nm/A]}
\text{j = inertia factor [Nms^2/rad]}
\text{d = drag (friction) [Nms/rad]}
\text{rm = motor resistance [Ohms]}
\text{lm = motor inductance [H]}
Electromagnetic Relay

Terminals

vopen: normally opened terminal [V,A]
vcomm: common terminal [V,A]
vclosed: normally closed terminal [V,A]
vctrl_n: negative control signal [V,A]
vctrl_p: 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

vtrig = input value to close relay [V]
vrelease = input value to open relay [V]
Three-Phase Motor

**Terminals**

- $vp_1, vn_1$: phase 1 terminals [V, A]
- $vp_2, vn_2$: phase 2 terminals [V, A]
- $vp_3, vn_3$: 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**

- $km = \text{motor constant [Vs/rad]}$
- $kf = \text{flux constant [Nm/A]}$
- $j = \text{inertia factor [Nms}^2\text{/rad]}$
- $d = \text{drag (friction) [Nms/rad]}$
- $rm = \text{motor resistance [Ohms]}$
- $lm = \text{motor inductance [H]}$
**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

sigin: (val, flow)
sigref: reference to which sigin is compared (val, flow)
sigout: comparator output (val, flow)

Description

Compares (sigin-sigin_offset) to sigref—the output is related to their difference by a tanh relationship.

If the difference >>> sigref, sigout is sigout_high.

If the difference = sigref, sigout is (sigout_high + sigout_low)/2.

If the difference <<< 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)

sigin_offset = subtracted from sigin 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

sigin: input (val, flow)
sigout: output (val, flow)

Description

Deadband region is when $sigin$ is between $sigin_{\text{dead high}}$ and $sigin_{\text{dead low}}$. $sigout$ is zero in the deadband region. Above the deadband, the output is $sigin - sigmoid_{\text{dead high}}$. Below the deadband, the output is $sigin - sigmoid_{\text{dead low}}$.

Instance Parameters

$sigin_{\text{dead high}} = \text{upper deadband limit (val)}$
$sigin_{\text{dead low}} = \text{lower deadband limit (val)}$
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

sigin_p, sigin_n: [V,A]
sigout_p, sigout_n: [V,A]

Description

val(sigout_p, sigout_n) = flow(sigin_p, sigin_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 (sigout_high + 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

sigin: (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 sigin minus sigin_offset unless the absolute value of the this is less than min_sigin. In this case, min_sigin is used as the adjusted input.

Instance Parameters

min_sigin = absolute value of minimum acceptable sigin (val)
gain = (val)
sigin_offset = input offset (val)
Multiplexer

Terminals

sigin1, sigin2, sigin3: 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 sigin1. If the differential-controlling signal is above sigth_low, sigout is sigin3. In between these two thresholds, sigout = sigin2.

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

sigin: (val, flow)
sigout: (val, flow)

Description

From 0 to period, sigout = sigin. After this, sigout is a periodic repetition of what sigin 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 \texttt{sigout\_max}, \texttt{sigin\_min}. \texttt{sigout0} must lie between \texttt{sigout\_max} and \texttt{sigin\_min}.

Instance Parameters

\texttt{sigout0 = initial sigout value (val)}

\texttt{gain = []}

\texttt{sigout\_max = maximum signal out (val)}

\texttt{sigout\_min = minimum signal out (val)}
Swept Sinusoidal Source

Terminals

\[ \text{sigout}_p, \text{sigout}_n: \text{output (val, flow)} \]

Description

The instantaneous frequency of the output is \( \text{sweep\_rate} \times \text{time} \) plus \( \text{start\_freq} \).

Instance Parameters

\[ \text{start\_freq} = \text{start frequency [Hz]} \]
\[ \text{sweep\_rate} = \text{rate of increase in frequency [Hz/s]} \]
\[ \text{amp} = \text{amplitude of output sinusoid (val)} \]
\[ \text{points\_per\_cycle} = \text{number of points in a cycle of the output []} \]
Three-Phase Source

Terminals

\texttt{vout\_a:} A-phase terminal \([V, A]\)
\texttt{vout\_b:} B-phase terminal \([V, A]\)
\texttt{vout\_c:} C-phase terminal \([V, A]\)
\texttt{vout\_star:} star terminal \([V, A]\)

Instance Parameters

\texttt{amp} = phase-to-phase voltage amplitude \([V]\)
\texttt{freq} = output frequency \([Hz]\)
Value-to-Flow Converter

Terminals

sigin_p, sigin_n: [V,A]
sigout_p, sigout_n: [V,A]

Description

flow(sigout_p, sigout_n) = val(sigin_p, sigin_n)

Instance Parameters

gain = value-to-flow gain []
Variable Frequency Sinusoidal Source

Terminals

sigin: frequency-controlling signal (val, flow)
sigout: (val, flow)

Description

Outputs a variable frequency sinusoidal signal. Its instantaneous frequency is \((\text{center}_\text{freq} + \text{freq}_\text{gain} \times \text{sigin}) \text{[Hz]}\)

Instance Parameters

amp = amplitude of the output signal (val)

center_freq = center frequency of oscillation frequency when sigin = 0 [Hz]

freq_gain = oscillator conversion gain (Hz/val)
Variable-Gain Differential Amplifier

Terminals

sigin_p, sighin_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, (sigctrl_p - sigctrl_n), and the adjusted input differential signal added to (sigout_high + sigout_low)/2. The adjusted input differential signal is the input differential signal minus signin_offset.

Instance Parameters

gain_const = amplifier gain when (sigctrl_p - 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 [m²]

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 [m²]
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 []
return = 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 [m²]
ms = saturation magnetization
gamma = shaping coefficient
k = bulk coupling coefficient
alpha = interdomain coupling coefficient
c = coefficient for reversible magnetization
Mathematical Components

Absolute Value

Terminals

sigin: (val, flow)
sigout: (val, flow)

Description

sigout is the absolute value of sigin.

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

sigin1, sigin2, sigin3, sigin4: (val, flow)
sigout: (val, flow)

Description

sigout = gain1*sigin1 + gain2*sigin2 + gain3*sigin3 + gain4*sigin4

Instance Parameters

gain1 = gain for sigin1 []
gain2 = gain for sigin2 []
gain3 = gain for sigin3 []
gain4 = gain for sigin4 []
Cube

Terminals
sigin: (val, flow)
sigout: (val, flow)

Description
sigout is the cube of the sigin.

Instance Parameters
None.
Cubic Root

Terminals

sigin:  (val, flow)

sigout:  (val, flow)

Description

sigout is the cubic root of sigin.

Instance Parameters

epsilon = small number added to sigin 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

sigin: (val, flow)
sigout: (val, flow)

Description

sigout is an exponential function of sigin. However, if sigin is greater than max_sigin, sigin is taken to be max_sigin. This is necessary because the exponential function explodes very quickly.

Instance Parameters

max_sigin = maximum value of sigin accepted (val)
Multiplier

Terminals

sigin1, sigin2: inputs (val, flow)  
sigout: terminals (val, flow)

Description

sigout = gain * sigin1 * signin2

Instance Parameters

gain = gain of multiplier []
Natural Log Function

Terminals

sigin: (val, flow)
sigout: (val, flow)

Description

sigout is the natural log of sigin, providing sigin > min_sigin. If sigin is between 0 and min_sigin, sigout is the log of min_sigin. If sigin is less than 0, an error is reported.

Instance Parameters

min_sigin = minimum value of sigin (val)
Polynomial

Terminals
sigin: (val, flow)
sigout: (val, flow)

Description
This is a model of a third-order polynomial function.

\[ \text{sigout} = p3 \times \text{sigin}^3 + p2 \times \text{sigin}^2 + p1 \times \text{sigin} + p0 \]

Instance Parameters

\[ p3 = \text{cubic coefficient} [] \]
\[ p2 = \text{square coefficient} [] \]
\[ p1 = \text{linear coefficient} [] \]
\[ p0 = \text{constant coefficient} [] \]
Power Function

Terminals

sigin:   (val, flow)
nigout: (val, flow)

Description

sigout is sigin to the power of exponent.

Instance Parameters

exponent = what sigin is raised by []

epsilon = small number added to sigin 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

sigin:  (val, flow)
sigout: (val, flow)

Description

This is a model of the sign of the input.

$\text{sigout is } +1 \text{ if } \text{sigin } \geq 0; \text{ otherwise, sigout is } -1.$

Instance Parameters

None.
Square

Terminals

sigin: input
sigout: output

Description

sigout is the square of the sigin.

Instance Parameters

None.
Square Root

Terminals
sigin: (val, flow)
sigout: (val, flow)

Description
sigout is the square root of sigin.

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

\(\text{sigin1, sigin2, sigin3, sigin4: (val, flow)}\)
\(\text{siginout: (val, flow)}\)

Description

\(\text{sigout} = \text{gain1*}\text{sigin1} - \text{gain2*}\text{sigin2} - \text{gain3*}\text{sigin3} - \text{gain4*}\text{sigin4}\)

Instance Parameters

\(\text{gain1 = gain for sigin1}\)
\(\text{gain2 = gain for sigin2}\)
\(\text{gain3 = gain for sigin3}\)
\(\text{gain4 = gain for sigin4}\)
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]
vclk: 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
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

\texttt{out\_pos, out\_neg: signal to measure} []

Description

This model is of a signal statistics probe.

This probe measures slope of a signal between \texttt{arg\_val1} and \texttt{arg\_val2}; if \texttt{arg\_val2} is not specified, it is set to the value exceeding \texttt{arg\_val1} by 0.1%.

Instance Parameters

\texttt{arg\_val1 = first argument value} []

\texttt{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

vamp_out: output voltage of opamp being measured [V,A]
vamp_p: positive terminal of opamp being measured [V,A]
vamp_n: negative terminal of opamp being measured [V,A]
vamp_spply_p: positive supply of opamp being measured [V,A]
vamp_spply_n: 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

vssplo_p = positive supply voltage required by opamp [V]
vssplo_n = negative supply voltage required by opamp [V]
tsettle = 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 sending 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]
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

sigin:  (val, flow)

Description

Samples $sigin$ 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

$tsample = \text{how often input is sampled [s]}$

$filename = \text{name of file where samples are stored []}$

$label = \text{label for signal being sampled []}$

$log\_time = \text{if the time variable should be logged to a file []}$
Slew Rate Measurement

Terminals

vamp_out: output voltage of the opamp being measured [V,A]
vamp_p: positive terminal of the opamp being measured [V,A]
vamp_n: negative terminal of the opamp being measured [V,A]
vamp_spply_p: positive supply of the opamp being measured [V,A]
vamp_spply_n: negative supply of the opamp being measured [V,A]

Description

Monitors the input and records the times at which it equals $v_{\text{start}}$ and $v_{\text{end}}$. The slew is given to be $v_{\text{start}} - v_{\text{end}}$ divided by the time difference.

The result is printed to the screen.

Instance Parameters

vspply_p = positive supply voltage required by opamp [V]
vspply_n = negative supply voltage required by opamp [V]
twait = time to wait before applying pulse to opamp input [V]
vstart = voltage at which to record the first measurement point [V]
vend = voltage at which to record the other measurement point [V]
tmin = 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
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

dosp, 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]  
vdo..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 +/- 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
tdel, trise, tfall = {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: $\frac{1}{N} \cdot \frac{(1 - Z^{-N})}{(1-Z^{-1})}$

Instance Parameters

$N = \text{oversampling ratio} [V]$

$v\text{trans}_\text{clk} = \text{transition voltage of the clock} [V]$

tdel, trise, tfall = {\text{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 +/-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

\( \text{vd0..vd7: data inputs [V,A]} \)

\( \text{vout: [V,A]} \)

Instance Parameters

\( \text{vref = reference voltage that conversion is with respect to [V]} \)

\( \text{vtrans = transition voltage between logic high and low [V]} \)

\( \text{tdel, trise, tfall = \{usual\} [s]} \)
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 Schockley 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]
\( v_{to} \) = threshold voltage [V]
gamma = bulk threshold []
\( \phi \) = bulk junction potential [V]
\( \lambda \) = channel length modulation []
\( \tau_{ox} \) = oxide thickness []
\( u_0 \) = transconductance factor []
\( x_j \) = metallurgical junction depth []
\( i_s \) = saturation current []
\( c_j \) = bulk junction capacitance [F]
\( v_j \) = bulk junction voltage [V]
\( m_j \) = bulk grading coefficient []
\( fc = \) forward bias capacitance factor []
\( \tau = \) parasitic diode factor []
\( cgbo = \) gate-bulk overlap capacitance [F]
\( cgso = \) gate-source overlap capacitance [F]
\( cgdo = \) 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]

`vsourc`: 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]`

`toxb = 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]
vsourc: 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]
ikr = reverse knee current [A]
cje = capacitance, base-emitter junction [F]
vje = voltage, base-emitter junction [V]
mje = b-e grading exponential factor []
cjc = capacitance, base-collector junction [F]
vjc = voltage, base-collector junction [V]
mjc = b-c grading exponential factor []
cjs = capacitance, collector-substrate junction [F]
vjs = voltage, collector-substrate junction [V]
mjs = c-s grading exponential factor []
fc = forward bias capacitance factor []
tf = ideal forward transit time [s]
xtf = tf bias coefficient []
vtf = tf-vbc dependence voltage [V]
itf = high current factor []
tr = reverse diffusion capacitance [s]
Schottky Diode

Terminals

\[ \text{vanode: anode voltage [V,A]} \]
\[ \text{vcathode: cathode voltage [V,A]} \]

Description

This model is of a diode based on the Schockley equation.

Instance Parameters

\[ \text{area = area of junction} \]
\[ \text{is = saturation current} \]
\[ \text{n = emission coefficient} \]
\[ \text{cjo = zero-bias junction capacitance [F]} \]
\[ \text{m = grading coefficient} \]
\[ \text{phi = body potential [V]} \]
\[ \text{fc = forward bias capacitance [F]} \]
\[ \text{tt = transit time [s]} \]
\[ \text{bv = reverse breakdown voltage [V]} \]
\[ \text{rs = series resistance [Ohms]} \]
\[ \text{gmin = minimal conductance [Mhos]} \]
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_lew_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:

\[ vout = unmod_amp \times (1 + mod_depth \times vin_adjusted) \times \cos(2 \times \pi \times f_carrier \times time) \]

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 \( t_{\text{start}} + \frac{t_{\text{period}}}{2} \) and every \( t_{\text{period}} \) 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

\( t_{\text{start}} \) = when to start measuring [s]

\( t_{\text{period}} \) = how often to compare bits [s]

\( v_{\text{trans}} \) = voltages above this at input are considered high [V]
Charge Pump

Terminals

vout: output terminal from which charge pumped/sucked [V,A]
vsrd: 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 of 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 $t_{period}$. During each decision period, a sample of the input is taken each $t_{sample}$. A count of the number of samples with values greater than $(v_{logic\_high} + v_{logic\_low})/2$ 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, $v_{logic\_low}$ is output. Otherwise, the output is $(v_{logic\_high} + v_{logic\_low})/2$.

The sampling starts at $t_{start}$.

Instance Parameters

t_{period} = period of binary data being extracted [s]
t_{sample} = sampling period [s]
v_{logic\_high} = output voltage for high [V]
v_{logic\_low} = output voltage for low [V]
t_{start} = time at which to start sampling [s]
t_{del}, t_{rise}, t_{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 change pump, a low-pass filter (LPF), and a digital voltage-controlled oscillator (VCO).

They are arranged in the following way:

```
Vin------| Phase |------| Charge |--->--|----------| |
        | Detector |------| Pump | ___|___ | |
        |___________| |________| | | |_______|
        || | R C | |
        | | |Network| |
        | | | (LPF) | |---Vout
        | V_local_osc | |_______|
        |-----------|
        |__|__|
gnd /////
```

**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:

center_freq + vco_gain * vin

Instance Parameters

center_freq = center frequency of oscillation frequency when vin = 0 [Hz]
vco_gain = oscillator conversion gain [Hz/volt]
vlogic_high = output voltage for high [V]
vlogic_low = output voltage for low [V]
tdel, trise, tfall = {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**

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vin_if</td>
<td>signal whose phase is being detected [V,A]</td>
</tr>
<tr>
<td>vin_lo</td>
<td>signal from local oscillator [V,A]</td>
</tr>
<tr>
<td>sigout_inc</td>
<td>logic signal to control charge pump [V,A]</td>
</tr>
<tr>
<td>sigout_dec</td>
<td>logic signal to control charge pump [V,A]</td>
</tr>
</tbody>
</table>

**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

\[ \text{vin}_1, \text{vin}_2: [V,A] \]
\[ \text{vout}: [V,A] \]

Description

\[ \text{vout} = \text{gain} \ast \text{vin}_1 \ast \text{vin}_2 \]

Instance Parameters

\[ \text{gain} = \text{gain of mixer} \]

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 \( t\_start + 0.5/\text{bit\_rate} \). 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 \( \text{vin\_min} \) and 255 representing \( \text{vin\_max} \). The first bit received is the LSB, bit 0; the last bit received is the MSB, bit 7.

The output rate is \( \text{bit\_rate}/8 \).

Instance Parameters

\( \text{freq\_sample} = \text{sample frequency [Hz]} \)

\( \text{t\_start} = \text{when to start sampling [s]} \)

\( \text{vout\_min} = \text{minimum input voltage [V]} \)

\( \text{vout\_max} = \text{maximum input voltage [V]} \)

\( \text{vtrans} = \text{voltages above this at input are considered high [V]} \)

\( \text{tdel, trise, tfall = \{usual\} [s]} \)
PCM Modulator, 8-Bit

**Terminals**

vin: input signal [V,A]

vout: modulated signal [V,A]

**Description**

The PCM modulator samples \( \text{vin} \) at a \( \text{sample_freq} \) [Hz] starting at \( \text{tstart} \). Once a sample has been obtained, it is converted to a linear 8-bit binary code with 0 representing \( \text{vin_min} \) and 255 representing \( \text{vin_max} \).

The bits are in the code and are sequentially put through \( \text{vout} \) at a rate 8 times \( \text{sample_freq} \) with \( \text{vlogic_high} \) signifying a 1 and \( \text{vlogic_low} \) 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 \( \text{vin_min} \) and \( \text{vin_max} \).

**Instance Parameters**

\( \text{sample_freq} = \text{sample frequency [Hz]} \)

\( \text{tstart} = \text{when to start sampling [s]} \)

\( \text{vin_min} = \text{minimum input voltage [V]} \)

\( \text{vin_max} = \text{maximum input voltage [V]} \)

\( \text{vlogic_high} = \text{output voltage for high [V]} \)

\( \text{vlogic_low} = \text{output voltage for low [V]} \)

\( \text{tdel, trise, tfall} = \{\text{usual}\} \) [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

\[ v_{out} = \text{amp} \times \sin(2 \times \pi \times f_{carrier} \times \text{time} + \text{phase}_{\text{max}} \times \text{vin}_{\text{adjusted}}) \]

where \( \text{vin}_{\text{adjusted}} \) is scaled version of \( \text{vin} \) that lies within the +/-1 range.

Before scaling, \( \text{vin} \) is limited to the range between \( \text{vin}_{\text{max}} \) and \( \text{vin}_{\text{min}} \) by clipping.

Instance Parameters

\( f_{\text{carrier}} = \) carrier frequency [Hz]

\( \text{amp} = \) amplitude of the PM modulator output [V]

\( \text{vin}_{\text{max}} = \) maximum acceptable input (clipping occurs above this) [V]

\( \text{vin}_{\text{min}} = \) minimum acceptable input (clipping occurs above this) [V]

\( \text{phase}_{\text{max}} = \) the phase shift produced when the modulating signal is at \( \text{vin}_{\text{max}} \) [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 \texttt{tstart}. \texttt{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 \texttt{vbox\_width} volts wide.

Instance Parameters

\texttt{freq} = demodulation frequency [Hz]
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

\[
\begin{array}{c|c|c|c|c}
\hline
\text{Q phase} & 0011 & 0111 & 1011 & 1111 \\
\text{i_phase} & 0010 & 0110 & 1010 & 1110 \\
\text{q_phase} & 0001 & 0101 & 1001 & 1101 \\
\text{v_trans} & 0000 & 0100 & 1000 & 1100 \\
\hline
\end{array}
\]

The two MSBs are encoded on the i-phase. The two LSBs are encoded on the q-phase.

The modulating formula is \( V_{out} = i_{phase} \cdot \cos(wt) + q_{phase} \cdot \sin(wt) \)

\( i_{phase} \) 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

\( \text{vin}_i, \text{vin}_q: \) quadrature inputs [V,A]
\( \text{vout}: \) modulator output [V,A]

Description

This takes two sampled quadrature inputs and does QPSK modulation on them.

Instance Parameters

\( \text{freq} = \) modulation frequency [Hz]
\( \text{amp} = \) modulator amplitude [V]
\( \text{vtrans} = \) voltages above this at input are considered high [V]
\( \text{tdel}, \text{trise}, \text{tfall} = \{\text{usual}\} \) [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

\( \text{vin: AM input signal [V,A]} \)
\( \text{vout: rectified AM signal [V,A]} \)

Description

\( \text{vin has } \text{noise_amp noise added to it and the resultant is attenuated by } \text{attenuation [dB]}. \)

Instance Parameters

\( \text{attenuation} = 20 \log_{10} \text{attenuation [dB]} \)
\( \text{noise_amp} = \text{amplitude of noise added to } \text{vin before } \text{attenuation [V]} \)
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]
Verilog-A Keywords

This appendix contains the list of the Cadence® Verilog®-A language keywords. *Keywords* are predefined nonescaped identifiers that are used to define the language constructs. Some keywords are not used in this release.

The simulator does not interpret a Verilog-A keyword preceded by a backslash character as a keyword. For more information, see “Identifiers” on page 46.

<table>
<thead>
<tr>
<th>keyword</th>
<th>keyword</th>
<th>keyword</th>
</tr>
</thead>
<tbody>
<tr>
<td>above</td>
<td>cmos</td>
<td>endtable</td>
</tr>
<tr>
<td>abs</td>
<td>connectrules</td>
<td>endtask</td>
</tr>
<tr>
<td>absdelay</td>
<td>cos</td>
<td>event</td>
</tr>
<tr>
<td>acos</td>
<td>cosh</td>
<td>exclude</td>
</tr>
<tr>
<td>acosh</td>
<td>cross</td>
<td>exp</td>
</tr>
<tr>
<td>ac_stim</td>
<td>ddt</td>
<td>final_step</td>
</tr>
<tr>
<td>aliasparam</td>
<td>ddx</td>
<td>flicker_noise</td>
</tr>
<tr>
<td>always</td>
<td>deassign</td>
<td>floor</td>
</tr>
<tr>
<td>analog</td>
<td>default</td>
<td>flow</td>
</tr>
<tr>
<td>analysis</td>
<td>defparam</td>
<td>for</td>
</tr>
<tr>
<td>and</td>
<td>delay</td>
<td>force</td>
</tr>
<tr>
<td>asin</td>
<td>disable</td>
<td>forever</td>
</tr>
<tr>
<td>asinh</td>
<td>discipline</td>
<td>fork</td>
</tr>
<tr>
<td>assign</td>
<td>discontinuity</td>
<td>from</td>
</tr>
<tr>
<td>atan</td>
<td>driver_update</td>
<td>function</td>
</tr>
<tr>
<td>atan2</td>
<td>edge</td>
<td>generate</td>
</tr>
<tr>
<td>atanh</td>
<td>else</td>
<td>genvar</td>
</tr>
<tr>
<td>begin</td>
<td>end</td>
<td>ground</td>
</tr>
<tr>
<td>bound_step</td>
<td>endcase</td>
<td>highz0</td>
</tr>
<tr>
<td>branch</td>
<td>endconnectrules</td>
<td>highz1</td>
</tr>
<tr>
<td>buf</td>
<td>enddiscipline</td>
<td>hypot</td>
</tr>
<tr>
<td>bufif0</td>
<td>endfunction</td>
<td>idt</td>
</tr>
<tr>
<td>bufif1</td>
<td>endmodule</td>
<td>idtmod</td>
</tr>
<tr>
<td>case</td>
<td>endnature</td>
<td>if</td>
</tr>
<tr>
<td>casex</td>
<td>endparamset</td>
<td>ifnone</td>
</tr>
<tr>
<td>casez</td>
<td>endprimitive</td>
<td>inf</td>
</tr>
<tr>
<td>ceil</td>
<td>endspecify</td>
<td>initial</td>
</tr>
<tr>
<td>initial_step</td>
<td>posedge</td>
<td>table_model</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>inout</td>
<td>potential</td>
<td>tan</td>
</tr>
<tr>
<td>input</td>
<td>pow</td>
<td>tanh</td>
</tr>
<tr>
<td>integer</td>
<td>primitive</td>
<td>task</td>
</tr>
<tr>
<td>join</td>
<td>pull0</td>
<td>temperature</td>
</tr>
<tr>
<td>laplace_nd</td>
<td>pull1</td>
<td>time</td>
</tr>
<tr>
<td>laplace_np</td>
<td>pullup</td>
<td>timer</td>
</tr>
<tr>
<td>laplace_zd</td>
<td>pulldown</td>
<td>tran</td>
</tr>
<tr>
<td>laplace_zp</td>
<td>pwr</td>
<td>tranif0</td>
</tr>
<tr>
<td>large</td>
<td>rcmos</td>
<td>tranif1</td>
</tr>
<tr>
<td>last_crossing</td>
<td>real</td>
<td>transition</td>
</tr>
<tr>
<td>limexp</td>
<td>realtime</td>
<td>tri</td>
</tr>
<tr>
<td>ln</td>
<td>reg</td>
<td>tril</td>
</tr>
<tr>
<td>localparam</td>
<td>release</td>
<td>triand</td>
</tr>
<tr>
<td>log</td>
<td>repeat</td>
<td>trior</td>
</tr>
<tr>
<td>macromodule</td>
<td>rnmos</td>
<td>trireg</td>
</tr>
<tr>
<td>max</td>
<td>rpmos</td>
<td>vectored</td>
</tr>
<tr>
<td>medium</td>
<td>rtran</td>
<td>vt</td>
</tr>
<tr>
<td>min</td>
<td>rtranif0</td>
<td>wait</td>
</tr>
<tr>
<td>module</td>
<td>rtranif1</td>
<td>wire</td>
</tr>
<tr>
<td>nand</td>
<td>scalared</td>
<td>wor</td>
</tr>
<tr>
<td>nature</td>
<td>sin</td>
<td>weak0</td>
</tr>
<tr>
<td>negedge</td>
<td>sinh</td>
<td>weak1</td>
</tr>
<tr>
<td>net_resolution</td>
<td>slew</td>
<td>while</td>
</tr>
<tr>
<td>nmos</td>
<td>small</td>
<td>white_noise</td>
</tr>
<tr>
<td>noise_table</td>
<td>specify</td>
<td>wire</td>
</tr>
<tr>
<td>nor</td>
<td>specparam</td>
<td>wreal</td>
</tr>
<tr>
<td>not</td>
<td>sqrt</td>
<td>wreal</td>
</tr>
<tr>
<td>notif0</td>
<td>strobe</td>
<td>xnor</td>
</tr>
<tr>
<td>notif1</td>
<td>strong0</td>
<td>xor</td>
</tr>
<tr>
<td>or</td>
<td>strong1</td>
<td>zi_pd</td>
</tr>
<tr>
<td>output</td>
<td>supply0</td>
<td>zi_np</td>
</tr>
<tr>
<td>parameter</td>
<td>supply1</td>
<td>zi_zd</td>
</tr>
<tr>
<td>pmos</td>
<td>table</td>
<td>zi_zp</td>
</tr>
</tbody>
</table>
Keywords to Support Backward Compatibility

The keywords in this section are provided for backward compatibility.

- abstol
- access
- bound_step
- ddt_nature
- delay
- discontinuity
- idt_nature
- temperature
- units
- vt
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 veriloga 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, vgnd) ;
 input vin, vgnd ;
 electrical vin, vgnd ;

 analog begin
  $strobe("%f, %f", xx, V(vin,vgnd)); // 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,vgnd));"
  "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...
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

This appendix explains how to set up a simulation of a circuit described in the Cadence® Verilog®-A language. For information, see the sections

- Creating a Verilog-A Module Description on page 478
- Creating a Spectre Netlist File on page 480
- Modifying Absolute Tolerances on page 483
- Using the Compiled C Code Flow on page 487

Except as noted, this appendix assumes that you are working outside of the Cadence analog design environment. For information on working inside the design environment, see Chapter 12, “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
```

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 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 196.

With the filename on the `include` directive, you can specify a full or relative path. As explained below, the path and filename that you specify control where the compiler searches for the file to be included.

File `res.va`, in the previous example, includes two files: `disciplines.vams` and `constants.vams`. These files are part of the Cadence distribution. 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.

The contents of the disciplines.vams and constants.vams files are listed in Appendix C, “Standard Definitions.” The files are located in the directory

your_install_dir/tools/spectre/etc/ahdl

where your_install_dir is the path to the Cadence installation directory.

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

`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.
The final relative path example illustrates how you might include a flip-flop module definition located in a sibling directory.

\`include "models/vloga/cap.va"

\`include "../logic/flip_flop.va"

**Simple Filename**

If you do not specify a path in the filename, the compiler searches the following three places, in the order given.

1. The directory that contains the file with the `\include` directive
2. The directory specified by the `CDS_VLOGA_INCLUDE` environment variable, if the variable is set
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. As a result, you generally do not have to worry about the location of these files.

If the file is not in any of the three places, the compiler issues an error message. If the file exists in more than one of these places, the first one encountered is included.

**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`
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 ahd1_include statement.

The netlist file res.ckt includes the Verilog-A description file res.va by using the ahd1_include statement. When the simulator encounters an ahd1_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 ahd1_include Spectre statement to include Verilog-A module description files in a netlist file. The ahd1_include statement has the form

```
ahdl_include "filename" [ -master mapped_name ]
```

If filename is not in the same directory as the netlist, you must ensure that filename either includes the complete path to the module file or is 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, modules that share the same name but are defined in different files can be
used in the same circuit. For example, the following two modules have the same name but different definitions.

The first module is defined in the file va_res.va.

```verilog
#include "discipline.h"
#include "constants.h"
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;
begin
  r=rsh*lr/(4.0*(wr-2*dw));
  V(plus, minus) <+ r*I(plus, minus);
end
endmodule
```

The next module is defined in the file another_res.va.

```verilog
#include "discipline.h"
#include "constants.h"
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;
begin
  r=rsh*lr/(8*(wr-4*dw));
  V(plus, minus) <+ r*I(plus, minus);
end
endmodule
```
To use both of these modules in a netlist, one of them is mapped to a different name with the -master option, as illustrated by the following netlist file. Notice how both the original res_va and the res_va that is mapped to the name res_va_mapped can be instantiated.

```
// 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
```

When you use Verilog-A within the Cadence analog design environment, the -master switch is inserted 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.

```verilog
// 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.

```verilog
// 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" // this will use `VOLTAGE_ABSTOL of 1e-7
`include "my_cap.va" // this will use `VOLTAGE_ABSTOL of 1e-7

module my_rc( in, out, gnd );
inout in, out, gnd;
electrical in, out, gnd;
parameter real r=1;
parameter real c=1n;
res #(r) r1 ( in, out );
cap #(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 ahd1_include statement.

```verbatim
// netlist file
// file "my_rc.ckt", rc_filter test circuit

global gnd
simulator lang=spectre
ahdl_include "my_rc.va"

// input voltage to filter
i1 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 214.

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.

Here is the netlist that instantiates the two modules.

```verbatim
// 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
i1 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**

```verbatim
// 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
```

December 2006

486

Product Version 6.1
File another_cap.va contains

```verbatim
// 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

This feature compiles the analog blocks of Verilog-A modules into shared objects for faster simulation. The resulting compiled objects can be saved so that all netlists that use the shared objects simulate faster. The shared objects can also be used by multiple designers so that the benefits of better performance extend beyond the person who compiles the modules. The following sections discuss the directories that are used and the environment variables that control the compiled C code flow.

### Turning the Compiled C Code Flow Off and On

Although using compiled C code is the default behavior of the tool, you can turn it off by either

- Setting the `CDS_AHDL_CMI_ENABLE` environment variable to `NO`.
  
  To resume using compiled C, either unset the `CDS_AHDL_CMI_ENABLE` variable or set the variable to `YES`.

- Setting the `spectre -ahdlcom` option to 0 (zero).
  
  The value 0 gives faster compilation (which, in this release, is achieved by turning off the compiled C code flow) but slower simulation. The value 1 gives slower compilation (because the compiled C code flow is turned on) but faster simulation.

  You can use `-ac` as shorthand for the `-ahdlcom` option.

By default, the compiled C code flow is on because the `CDS_AHDL_CMI_ENABLE` variable is unset (with a default value of `YES`) and the `ahdlcom` option is unset (with a default value of 1). If these defaults do not meet your needs, you can use the information in the following sections to set different values.
Creating and Specifying Compiled C Code Databases

The compiled C code flow stores the shared objects in a database on disk for the simulation to use. The shared objects are stored in a directory termed the ahdlSimDB (AHDL simulation database). By default, this database is created in the current working directory and given a name created by appending .ahdlSimDB to the root of the circuit name. For example, if the circuit name is top4.sys, the database name might be top4.ahdlSimDB.

You can specify an alternative location for the ahdlSimDB by setting the CDS_AHDLCMI_SIMDB_DIR environment variable to the path of a directory. If the path is writable, the ahdlSimDB is created there. If the path is not writable or does not exist, an error is reported.

To store compiled objects, you use a second type of database, termed ahdlShipDBs (AHDL ship databases). To create such databases, you set the CDS_AHDLCMI_SHIPDB_COPY environment variable to YES. When you use this setting, an ahdlShipDB for each Verilog-A file is created in the directory that contains the Verilog-A file, if the directory is writable. If the directory is not writable, no ahdlShipDBs are created for the modules in the Verilog-A file that is being processed.

If the CDS_AHDLCMI_SHIPDB_DIR environment variable (or the equivalent, but obsolete CDS_AHDLCMI_DIR variable) is also set to a writable path, the ahdlShipDB database is created there and shared by all the Verilog-A files used for simulations that are run while this environmental variable is set. If the CDS_AHDLCMI_SHIPDB_DIR variable is not set to a writable path or the path does not exist, a warning is reported and ahdlShipDBs are not created.

While looking for already compiled shared objects, Spectre automatically looks for ahdlShipDBs in the same location as the Verilog-A files. If CDS_AHDLCMI_SHIPDB_DIR is set 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 shared objects stored in the ahdlSimDB are reused automatically.

To minimize the compilation of shared objects when you run different netlists that share the same Verilog-A files, do one of the following:

- Set the CDS_AHDLCMI_SHIPDB_COPY environment variable to YES. Shared objects generated by the first simulation are put in the ahdlShipDB created for each Verilog-A file in the same directory as the Verilog-A file being processed. The shared objects are reused by subsequent simulations. This approach works if the Verilog-A files are in directories that are writable.
Set the `CDS_AHDLCMI_SHIPDB_COPY` environment variable to YES and set `CDS_AHDLCMI_SHIPDB_DIR` to a writable directory. This directory becomes the sole ahdlShipDB. Shared objects generated by the first simulation are put in this ahdlShipDB. The shared objects are re-used by subsequent simulations. This approach works if the Verilog-A files are in directories that are read-only.

To share precompiled objects among different users,

- Run the simulation once with the `CDS_AHDLCMI_SHIPDB_COPY` variable set to YES.

  An ahdlShipDB is created 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 ahdlShipDBs contain shared objects.

  Other users who reference 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.
Supported and Unsupported Language Elements


The Cadence implementation of Verilog-A does not support the following elements of the specified Verilog-A language.

- The following two aspects of hierarchy:
  - Ordered parameter lists in hierarchical instantiation
  - Named nodes in hierarchical instantiation
- Hierarchical names, except for `node.potential.abstol` and `node.flow.abstol`, which are supported
- Derived natures
- Using `1'b1` constant specification
- Parameters used to specify ranges for the `generate` statement
- String values used in parameter arrays
- The `defparam` statement
- The `ground` declaration
- Nested use of the `ddt` operator
- Module description attribute
- Environment parameter functions (`$simparam`)
- Hierarchy detection functions (`$param_given`, `$port_connected`)
- Predefined macros (`VAMS_COMPACT_MODELING`)
Local parameter declarations
- Limiting functions
- Limiting algorithms
- String parameter ranges
- The following four aspects of functions:
  - Arrays passed to functions
  - Nodes passed to functions
  - Access functions used inside functions
  - Accessing variables defined in a function’s parent module
- The following aspect of input and output:
  - The %b format character
- Vector branches
- Vector arguments for simulator functions
- The concatenation operator
- The derivative operator
- Laplace transforms taking parameter-sized arrays as arguments
- Parameter-sized ports
- Enforcement of input, output, and inout
- The following system tasks:
  - $\text{realtime scaled to the `timescale directive}$
  - $\text{stime}$
  - $\text{time}$
  - $\text{monitor and fmonitor}$
  - The %b, %o, and %h specifications for $\text{display, fdisplay, write, fwrite,}$
    $\text{monitor, fmonitor, strobe, and fstrobe}$
  - $\text{monitor off/on}$
  - $\text{printtimescale}$
$timeformat
$blestoreal
$itor
$real2bits
$rtoi

$readmen used with the %b, %h, and %r specifications.

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

<table>
<thead>
<tr>
<th>Deprecated feature</th>
<th>To comply with the current standard,...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ifdef</code></td>
<td>Use <code>ifdef</code> without a trailing tick. For example, instead of <code>ifdef</code>CHECK_BACK_SURFACE use <code>ifdef</code> CHECK_BACK_SURFACE</td>
</tr>
<tr>
<td><code>inf</code> used for specifications other than ranges</td>
<td>Use <code>inf</code> only to specify ranges.</td>
</tr>
<tr>
<td>user-defined analog function</td>
<td>Use analog function</td>
</tr>
<tr>
<td>discontinuity</td>
<td>Use $discontinuity.</td>
</tr>
<tr>
<td>I(a,a) to probe a port current</td>
<td>Use I().</td>
</tr>
<tr>
<td>delay</td>
<td>Use absdelay.</td>
</tr>
<tr>
<td>Null statements used elsewhere other than in case and event statements</td>
<td>Use null statements (coded as ; ) only in case or event control statements.</td>
</tr>
<tr>
<td>Chained assignment statements, such as x=y=z</td>
<td>Break the assignment chain into separate assignments, such as y=z; x=y;.</td>
</tr>
<tr>
<td>$limexp</td>
<td>Use $limexp.</td>
</tr>
<tr>
<td>Using [ ] for literal arrays</td>
<td>Use {} for literal arrays.</td>
</tr>
<tr>
<td>bound_step</td>
<td>$bound_step</td>
</tr>
</tbody>
</table>
### Deprecated features

<table>
<thead>
<tr>
<th>Deprecated feature</th>
<th>To comply with the current standard,...</th>
</tr>
</thead>
<tbody>
<tr>
<td>export qualifier</td>
<td>Delete the export qualifier, which is redundant.</td>
</tr>
<tr>
<td>$dist_ functions in the analog block</td>
<td>Use the corresponding $rdist_ function.</td>
</tr>
<tr>
<td>The second argument of the cross operator being a non-integer type</td>
<td>Change the second operator to an integer type.</td>
</tr>
<tr>
<td>Using for, while and repeat loop statements for the timer function</td>
<td>Use a genvar loop for the timer function.</td>
</tr>
<tr>
<td>Unassigned variables</td>
<td>Assign each variable. Unassigned variables are considered digital variables.</td>
</tr>
<tr>
<td>generate</td>
<td>Use a genvar loop instead.</td>
</tr>
<tr>
<td>The second argument of the last_crossing operator being a non-integer type</td>
<td>Change the second operator to an integer type.</td>
</tr>
</tbody>
</table>

The items in the next list are Cadence extensions. These features are not part of the standard specification of the language.

### Cadence extensions

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cadence syntax for attributes</td>
</tr>
<tr>
<td>mfactor attribute</td>
</tr>
<tr>
<td>dynamicparams</td>
</tr>
<tr>
<td>Inherited parameters</td>
</tr>
</tbody>
</table>
# Updating Verilog-A Modules

The Verilog-A language is a subset of Verilog-AMS, but some of the language elements in that subset have changed since Verilog-A was released by itself. As a consequence, you might need to revise your Verilog-A modules before using them as Verilog-AMS modules. The following table highlights the differences.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Independent Verilog-A</th>
<th>Verilog-AMS</th>
<th>Change type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog time</td>
<td>$realtime</td>
<td>$abstime</td>
<td>New</td>
</tr>
<tr>
<td>Empty discipline</td>
<td>Predefined as type wire</td>
<td>Type not defined</td>
<td>Default definition</td>
</tr>
<tr>
<td>Implicit nodes</td>
<td><code>default_nodetype discipline_identifier default: wire</code></td>
<td>default type: empty discipline, no domain type</td>
<td>Default definition</td>
</tr>
<tr>
<td>initial_step</td>
<td>Default = TRAN</td>
<td>Default = ALL</td>
<td>Default definition</td>
</tr>
<tr>
<td>final_step</td>
<td>Default = TRAN</td>
<td>Default = ALL</td>
<td>Default definition</td>
</tr>
<tr>
<td>$realtime</td>
<td>$realtime: timescale =1 sec</td>
<td>$realtime: timescale=`timescale def=1n. See $abstime</td>
<td>Definition</td>
</tr>
<tr>
<td>Discontinuity function</td>
<td>discontinuity(x)</td>
<td>$discontinuity(x)</td>
<td>Syntax</td>
</tr>
<tr>
<td>Limiting exponential function</td>
<td>$limexp(expression)</td>
<td>limexp(expression)</td>
<td>Syntax</td>
</tr>
<tr>
<td>Port branch access</td>
<td>I(a,a)</td>
<td>I(&lt;a&gt;)</td>
<td>Syntax</td>
</tr>
</tbody>
</table>

**Note:** Cadence® Verilog-A supports only this form.

**Note:** This form is not supported in Cadence Verilog-A.
Suggestions for Updating Models

The remainder of this appendix describes some of these changes in greater detail and suggests ways of modifying your existing Verilog-A models so that they work in version 4.4.6 of Verilog-A and in version 1.0 of Verilog-AMS. The changes recommended here might not work with 4.4.5 or earlier versions of Verilog-A.

Current Probes

OVI Verilog-A 1.0 syntax for a current probe is `I(a,a)`. OVI Verilog-AMS 2.0 changes this to `I(<a>)`.

**Suggested change:** Put `I(<a>)` inside an `ifdef __VAMS_ENABLE__`, which makes the syntax effective only for Verilog-AMS. For example, change

```verilog
iin_val = I(vin,vin);
```

---

<table>
<thead>
<tr>
<th>Feature</th>
<th>Independent Verilog-A</th>
<th>Verilog-AMS</th>
<th>Change type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timestep control (maximum stepsize)</td>
<td><code>bound_step(const_expression)</code></td>
<td><code>$bound_step(expr)</code></td>
<td>Syntax</td>
</tr>
<tr>
<td>Continuous waveform delay</td>
<td><code>delay()</code></td>
<td><code>absdelay()</code></td>
<td>Syntax</td>
</tr>
<tr>
<td>User-defined analog functions</td>
<td>Function</td>
<td>Analog function</td>
<td>Syntax</td>
</tr>
<tr>
<td>Discipline domain</td>
<td>N/A, assumed continuous</td>
<td>Now continuous (default) and discrete</td>
<td>Extension</td>
</tr>
<tr>
<td>Time tolerance on timer functions</td>
<td>N/A</td>
<td>Supports additional time tolerance argument for <code>timer()</code></td>
<td>Extension</td>
</tr>
<tr>
<td><code>default_nodetype</code></td>
<td><code>default_nodetype</code></td>
<td><code>default_discipline</code></td>
<td>Obsolete</td>
</tr>
<tr>
<td>Generate statement</td>
<td><code>generate</code></td>
<td><code>genvar</code></td>
<td>Obsolete</td>
</tr>
<tr>
<td>Null statement</td>
<td>;</td>
<td>Limited to case, conditional, and event statements</td>
<td>Obsolete</td>
</tr>
</tbody>
</table>
to
`ifdef __VAMS_ENABLE__
iin_val = I(<vin>);
`else
  iin_val = I(vin,vin);
`endif

Verilog-A warning: None

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: None

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: None
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;

```verilog
parameter real points_per_cycle = inf from [6:inf];
```

to

```verilog
parameter real points_per_cycle = 1M from [6:inf];
```

Verilog-A warning: None

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 warning: None

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 warning: None
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

```verilog
svcvs #(poles([-2*PI*bw,0])) output_filter
```

to

```verilog
svcvs #(poles({-2*PI*bw,0})) output_filter
```

**Verilog-A warning:** None

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

```verilog
x=y=z;
```

to

```verilog
y = z; x = y;
```

**Verilog-A warning:** None

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

```verilog
@(cross(V(in),1.0) begin
```

to

```verilog
@(cross(V(in),1) begin
```

**Verilog-A warning:** None

$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 warning: None

`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: None

$warning is Not Allowed

Spectre-Verilog-A supports $warning, but OVI Verilog-AMS 2.0, 1.0 and 1364 do not support this as a standard built-in function.

Suggested change: Change $warning to $strobe.

Verilog-A warning: None

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 warning: None
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 tl_cell [?view tl_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

\[ t_lib \]
Name of the library to be updated.

\[ tl_cell \]
Name or list of names of cells to be updated. If \( tl_cell \) is omitted, the function updates every veriloga cellview in the library.

\[ tl_view \]
Name or list of names of cellviews to be updated. If \( tl_view \) 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.
Glossary

A

analog HDL
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 $ddt$ 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.
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 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

! (logical negation) 89
!= (not equal to) 91
!=(inequality comparison string function) 99
- (binary minus) 91
- (unary minus) 89
" (double quote character), displaying 166
\$ (dollar sign), in identifiers 47
\$abstime function 123
\$display 169
\$display task 169
\$dist_chi_square function 136
\$dist_erlang function 138
\$dist_exponential function 135
\$dist_normal function 134
\$dist_poisson function 136
\$dist_t function 137
\$dist_uniform function 133
\$fclose task 176
\$fdisplay 175
\$fdisplay task 175
\$fopen task 171
  special formatting commands for 172
\$fstrobe 174
\$fstrobe task 174
\$fwrite 175
\$limexp
  analog operator 144
  changed to limexp 499
\$limit function 121
\$random simulator function 132
\$realtime function 123
\$realtime to \$abstime 498
\$sscanf string function 100
\$strobe 166
  description 166, 170
  example of use 168
\$warning, not allowed 500
\$write 169
% (modulo) 91
% (percent character), displaying 166
& (bitwise binary and) 92
&& (logical and) 92
( (left parenthesis) 57
) (right parenthesis) 57
* (multiply) 91
+ (binary plus) 90
+ (unary plus) 89
.va file extension 478
/ (divide) 91
/* (slash, asterisk), as comment marker 46
// (double slash), as comment marker 46
/f 504
< (less than) 91
++ (branch contribution operator) 76
<< (shift bits left) 92
<= (less than or equal) 91
== (logical equals) 91
> (greater than) 91
>= (greater than or equal) 91
>> (shift bits right) 92
? and : (conditional operator) 94
@ (at-sign) operator 110
[ (left bracket), using to include end point in range 57
\ (backslash)
  continuing macro text with 194
  displaying 166
  in escaped names 47
] (right bracket), using to include end point in range 57
^ (bitwise binary exclusive OR) 92
^\ (bitwise binary exclusive NOR) 92
_(underscore), in identifiers 47
\ (accent grave) 194
`define compiler directive 194
  modifying abstol with 484
  syntax 194
  tested by `ifdef compiler directive 196
`if `MACRO, not allowed 500
`ifdef compiler directive 196
`include compiler directive 196
`resetall compiler directive 197
`timescale compiler directive 197
`undef compiler directive 196
| (bitwise binary or) 92
|| (logical or) 92
~ (bitwise unary negation) 89
~\ (bitwise binary exclusive nor) 92
^ (bitwise binary exclusive OR) 92
timer 115
analog functions 497
analog multiplexer 293
analog multiplexer model 293
analog operators 143
$limexp 144
not allowed in for loop 82
listed 143
not allowed in repeat loop 81
restrictions on 144
using in looping constructs 83
not allowed in while loop 82
analog systems 26
analog-to-digital converter
example 84
model, 8-bit 420
model, 8-bit (ideal) 421
model, 8-bit differential nonlinearity measurement 393
model, 8-bit integral nonlinearity measurement 394
analyses
detecting first time step in 111
detecting last time step in 111
analysis function 128
analysis types 128
analysis-dependent functions 128
AND gate 326
AND gate model 326
angular velocity 264, 265
arc-cosine function 107
arc-hyperbolic cosine function 107
arc-hyperbolic sine function 107
arc-hyperbolic tangent function 107
arc-sine function 107
arc-tangent function 107
arc-tangent of x/y function 107
array specifications, changing 499
arrays
arguments represented as 156
as parameter values 188
assignment operator for 76
of integers, declaring 52
of parameters 58
of reals, declaring 53
ASCII code, returning from character 96
asin function 107
asinh function 107
assignment operator, procedural 76
assignment statement 75
assignment statement, indirect branch 78
associated reference directions 27
association order, of operators 88
atan function 107
atan2 function 107
atanh function 107
atoi function
details 100
atoi operator 96
atoi string function 100
atoreal function 96
details 101
atoreal string function 101
attenuator model 442
attributes
abstol 64
access 64
accessing 127
blowup 65
ddt_nature 64
huge 65
idt_nature 64
no_rigid_switch_branch 279
requirements 65
units 64
user-defined 64
using to define base nature 64
audio source 443
audio source model 443

B
backward compatibility 473
base natures
declaring 64
description 63
basic components 310
behavioral characteristics, defining with internal nodes 42
behavioral description, definition 503
behavioral model, definition 503
bidirectional ports 35
binary operators 90
binding, run-time, definition 507
bit error rate calculator model 444
bitwise operators 93
AND 93
exclusive NOR 93
exclusive OR 93
inclusive OR 93
clamp model
hard current  295
hard voltage  296
soft current  301
soft voltage  302
clocked JK flip-flop model  334
closing a file  176
code generator model
2-bit  446
4-bit  447
comments  46
  in modules  46
  in text macros  194
comparator  350
  example  152
  model  350
comparison operator
details  99
comparison operators, for strings  96, 99
compatibility
  of disciplines  67
  node connection requirements  185
  of disciplines  67
compensator model
  lag  319
  lead  320
  lead-lag  321
compilation, conditional  196
compiler directives
  `define  194
  ifdef  196
  `include  196, 197
  `resetall  197
  `timescale  197
  `undef  196
designated by accent grave (`)  194
list of  194
resetting to default values  197
using  194
compiling code conditionally  196
components
  creating multiple cellviews for  217
definition  503
concatenation operator  99
details  99
concatenation operator, for strings  96
conditional compilation  196
conditional operator  94
conditional statement  80
configuration
  needed for multiple cellviews  219
opening in Cadence analog design
  environment  219, 225
connecting instances
  example  184
  rules for  185
connecting the ports of module
  instances  184
conservative discipline  67
conservative systems  27
  conservative disciplines used to define  72
defined  27
  values associated with  27
constant expression  88
constant power sink model  299
constants
  integer  48
  real  48
  standard  289
  string, used as parameters  189
constants.vams file  288
  contents of  283
  location of  479, 480
  role in simulation  479
constitutive equations  266
constitutive relationships
definition  270, 503
use in nodal analysis  271
constructs
  case  80
  looping  83
  procedural control  75
contribution statements, format  38, 76
control components  318
control flow
definition  504
describing behavior with  38
controlled integrator model  351
controlled sources  276
controller model
  proportional  322
  proportional derivative  323
  proportional integral  324
  proportional integral derivative  325
conventions, typographical  20
convergence  271
conversion specifications  167
converting real numbers to integers  53
copy operator, for strings  96
core model, magnetic  372
cos function  107
cosh function  107
cosine function  107
Create New File form  210
cross event  112
cross function
  syntax  112
cube model  379
cubic root model  380
current analysis type, determining  128
current clamp model
  hard  295
  soft  301
current deadband amplifier model  294
current meter model  395
current probes  496
current source model
  current-controlled  316
  voltage-controlled  315
current-controlled current source
  277, 316
current-controlled voltage source
  model  316
current-controlled voltage source
  model  314

d
D
  d or D format character  167
DAC model
  8-bit  423
  8-bit (ideal)  424
  8-bit differential nonlinearity
    measurement  396
  8-bit integral nonlinearity
    measurement  397
DAC, definition  396
damper model  414
data types
  branch  72
  discipline  66
  integer number  52
  nature  63
  parameter  54
  real number  52
DC analysis
  value returned by idt during  146
DC motor model  346
ddt operator (time derivative)  40, 144
ddt_nature attribute
  description  64
  requirements for  65
deadband amplifier model
  current  294
  voltage  308
  deadband differential amplifier model  353
deadband model  352
deriv model  448
decimal logarithm function  106
decimator model  422
declarations
  definition  504
  global, definition  504
  .def filename extension  291
default values, required for parameters  56
define compiler directive
  modifying abstol with  484
  syntax  194
delay operator  150
delay to absdelay  498
delaying continuously valued waveform  150
deleting parameters from a veriloga or ahdl
  Cellview  221
delta probe model  398
demodulator model
  8-bit PCM  456
  AM  440
  FM  451
  PM  460
  QAM 16-ary  462
  QPSK  465
derivative controller model
  proportional  323
  proportional integral  325
  derivative, time  144
derived nature  63
descend dialog  226
descend edit  209
descend edit command  209
describing a system  25
description attribute
  for integers  52
  for net disciplines  70
  for parameter declarations  55
  for reals  53
differential amplifier (opamp)  354
differential amplifier model  354
  deadband  353
  limiting  361
  variable gain  371
differential signal driver  355
differential signal driver model 355
differentiator model 356
digital phase locked loop model 449
digital to analog converter example 155
digital voltage controlled oscillator model 450
digital-to-analog converter model
  8-bit 423
  8-bit (ideal) 424
  8-bit differential nonlinearity measurement 396
  8-bit integral nonlinearity measurement 397
diode model 432
  Schottky 439
direction of ports, declaring 35
directions, reference 507
directives. See compiler directives
disciplines 66
  compatibility of 67 to 70
  conservative 67
  declaring 66
  definition 504
  empty 67
  empty, declaring terminals with 71
  scope of 66
  signal-flow 67
disciplines.vams file 284
  contents of 283
  location of 479, 480
  required in Cadence analog design environment 486
  role in simulation 478
discontinuities
  announcing 119
  in switch branches 278
discontinuity function
  changed to $discontinuity 500
  not required for switch branches 278
  syntax 119
discrete-time finite difference approximation 271
$display task 169
displaying
  information 165
  results 165
  waveforms of variables 240
$dist_chi_square function 136
$dist_erlang function 138
$dist_exponential function 135
$dist_normal function 134
$dist_poisson function 136
$dist_t function 137
$dist_uniform function 133
distributions
  chi-square 136
  Erlang 138
  exponential 135
  gaussian 135
  normal 134
  Poisson 136
  Student's T 137
  uniform 133
divider model 381
DNL, definition 393
dollar signs, in identifiers 47
domain
  of hyperbolic functions 107
  of mathematical functions 106
  of trigonometric functions 107
driver model
differential signal 355
D-type flip-flop model 333
dynamic expression, definition 504

E
E 504
e or E format character 167
Edit Object Properties form 227
8-bit parallel register model 344
8-bit serial register model 345
electrical modeling 243
electromagnetic components 346
electromagnetic relay 347
electromagnetic relay model 347
element, definition 504
else statement, matching with if statement 80
empty disciplines
  compatibility of 68
  definition 67
  example 67
  predefined (wire) 67
endmodule keyword 32
entering interactive Tcl mode 177
enumerated values, as parameter values 189
environment functions 123
environment variables
  CDS_VLOGA_INCLUDE 480
equality comparison string function (==) 99
Erlang distribution 138
Erlang distribution function 138
error calculation block 318
error calculation block model 318
error messages, forms of 475
escaped names 47
defined 47
in Cadence analog design
environment 214
Spectre 47
using in the Cadence analog design
environment 214
using keywords as 471
event OR operator 110
events
detecting analog 110
detecting and using 110
events, analog 109 to 115
examples
$strobe formatting 168
analog-to-digital converter 84
car 262
gearbox 266
ideal relay 278
ideal sampled data integrator 164
inductor 40
limiter 258
linear damper 257
motor 248
rectifier 243
RLC circuit 42
road 258
shock absorber 257
sources and probes 280
spring 256
thin-film transistor 249
thyristors 245
transformer 246
voltage deadband amplifier 39
wheel 260
exclude keyword 57
exiting to the operating system 176
exp function 106
exponential distribution function 135
exponential function 106
exponential function model 382
exponential function, limited 144
expressions
constant 88
definition 88
dynamic, definition 504
short circuiting of 95
F
F 504
f or F format character 167
fault model
open circuit 297
short circuit 300
$fclose task 176
$fdisplay task 175
file extension .va 478
files
closing 176
including at compilation time 196
opening 171
writing to 174
files, working with 171
filters
slew 154
transition 151
final_step event 111
find event probe 399
find event probe model 399
find slope 401
find slope model 401
finite-difference approximation 271
flicker_noise function 131
flicker_noise simulator function 131
flip-flop model
clocked JK 334
D-type 333
JK-type 336
RS-type 338
toggle-type 339
trigger-type 339
flow
default value for 277
definition 504
in a conservative system 27
probes, definition 274
probes, in port branches 274
sources, definition 275
sources, equivalent circuit model
for 276
sources, switching to potential
sources 277
flow law. See Kirchhoff's Laws, Flow Law
flow-to-value converter model 357
FM demodulator 451
FM demodulator model 451
FM modulator model 452
$fopen task 171
for loop statement 82
for statement 82
formatting output 167
forms
  Add Block 205
  Cellview From Cellview 207, 213, 218
  Cellviews Need Saving 224
  Create New File 210
  Edit Object Properties 227
  New Library 203
  Open Configuration or Top Cellview 219
  Simulation Environment Options 238
  Symbol Generation 212
  Technology File for New Library 204
four-number adder model 378
four-number subtractor model 392
freeform block shape 206
frequency meter model 402
frequency-phase detector model 453
$fstrobe task 174
full adder model 341
full subtractor model 343
full wave rectifier model, two phase 429
functions
  access 126
  defining 177
  environment 123
  mathematical 105
  string 95, 96
  user-defined 177

G
G 504
  g or G format character 167
  gain block 182
  gap model, magnetic 373
  gate pulses, used to control thyristors 245
  gaussian distribution 135
  gearbox
    behavioral description for 266
    model 263
    netlist for 267
  gearbox model 413
  generate statement 83
  generating random numbers 132
  generating random numbers in specified distributions 133
  genvars 62
  getc function 96
  getc string function 101
  global declarations, definition 504
  ground nodes
    as assumed branch terminal 73
    compatibility of 185
    potential of 27

H
H 505
  h or H format character 167
  half adder model 340
  half subtractor model 342
  half wave rectifier model, two phase 430
  hard current clamp model 295
  hard voltage clamp model 296
  HDLdebug debugger 19
  hierarchical module instantiation 235
  hierarchical name, displaying 166
  hierarchical Verilog-A modules 235
  Hierarchy Editor
    synchronizing with schematic 224
    window 221
  hierarchy, using 237
  higher order systems 42
  huge attribute, description 65
  hyperbolic cosine function 107
  hyperbolic functions 106
  hyperbolic sine function 107
  hyperbolic tangent function 107
  hypot function 107
  hypotenuse function 107
  hypotenuse function 107
  hysteresis model, rectangular 358

I
IC analysis, value returned by idt during 146
  ideal relay example 278
  ideal sampled data integrator example 164
  identifiers 46
  idt operator
    example 41
using in feedback configuration 146
idt_nature attribute
description 64
requirements for 65
idtmod operator
example 148
using 147
`ifdef compiler directive 196
ignored code, restrictions on 196
impedance meter model 412
implicit branches 73
implicit models 261
include Compiler Directive 478
`include compiler directive 196
including files
at compilation time 196
in a netlist 481
including Verilog-A through model
setup 235
indirect branch assignment statement 78
inductor model 312
module describing 40
untrimmed 306
inequality comparison string function
(!=) 99
inertia 265
-inf (negative infinity) 57
inf used as a number 498
infinity, indicating in a range 57
inh_conn_def_value attribute 189
inh_conn_prop_name attribute 189
inherited connections, attributes for 189
inherited parameters, attribute for 55
inherited ports, using 189
inherited_mfactor attribute 191
initial_step event 111
example of use 258
syntax 111
instance parameters, modifying 230
instances
associating cellviews with 222
connecting with ports 184
creating 182
creating and naming 182
definition 505
examining Verilog-A modules bound
to 226
labels for, in Cadence analog design
environment 206
overriding parameter values in 185
instantiating
analog primitives 186
analog primitives that use array valued
parameters 188
module description files in netlists 480
modules that use unsupported
parameter types 189
modules with netlists 43
Verilog-A modules 182
instantiation
definition 505
hierarchical 480
of non-Verilog-A modules 189
statement. See module instantiation
statement example
syntax 182
integer
attributes for 52, 191, 279
constants 48
data type 52
declaring 52
numbers 48, 52
range allowed in Verilog-A 52
integral controller model, proportional 324
integral derivative controller model,
proportional 325
integral, time 145
integration and differentiation with analog
signal, using 40
integrator 359
integrator model 359
controlled 351
saturating 366
switched capacitor 428
interconnection relationships 270
interface declarations, example 33
internal nodes
for higher order derivatives 40
in higher order systems 42
use 41
internal nodes in behavioral definitions,
using 41
internal nodes in higher order system,
using 42
internal nodes in modules, using 41
internet mail address 19
interpolating with table models 139

J
JK-type flip-flop model 336
K

keywords, list of 471
Kirchhoff's Laws 270
  definition 505
  Flow Law 27, 270, 271, 272
  illustrated 270
  use in nodal analysis 271
  Potential Law 27, 270

L

L 505
lag compensator model 319
Laplace transforms
  numerator-denominator form 159
  numerator-pole form 158
  s-domain filters 156
  zero-denominator form 157
  zero-pole form 157
laplace_nd Laplace transform 159
laplace_np Laplace transform 158
laplace_zd Laplace transform 157
laplace_zp Laplace transform 157
last_crossing simulator function
  improving accuracy of 122
  setting direction for 112, 122
  syntax 122
laws, Kirchhoff's. See Kirchhoff's Laws
lead compensator model 320
lead-lag compensator model 321
left justifying output 167
len function 96
len string function 101
level shifter model 337, 360
level, definition 505
libraries
  creating 202
$limexp analog operator 144
limited exponential function 144
limiter model 258
limiting differential amplifier model 361
linear conductor model 280
linear damper model 257
linear resistor model 281
In function 106
local parameters
  declaring 59
log function 106

logarithm function
  decimal 106
  natural 106
logarithmic amplifier model 362
logic components 326
logic table 336, 338, 339
lowercase characters, required for SPICE-mode netlisting 483
LPF, definition 449

M

M 505
.m suffix, required for models 240
M_1_PI constant 289
M_2_PI constant 289
M_2_SQRTPI constant 289
M_E constant 289
M_LN10 constant 289
M_LN2 constant 289
M_LOG10E constant 289
M_LOG2E constant 289
M_PI constant 289
M_PI_2 constant 289
M_PI_4 constant 289
M_SQRT1_2 constant 289
M_SQRT2 constant 289
M_TWO_PI constant 289
macros. See text macros
magnetic components 372
magnetic core 372
magnetic core model 372
magnetic gap 373
magnetic gap model 373
magnetic winding 374
magnetic winding model 374
mapping instance ports to module
  ports 183
mapping ports with ordered lists 183
mass model 415
math domain errors, controlling 107
mathematical components 376
mathematical functions 106
maximum (max) function 106
measure components 393
measurement model
  offset 403
  slew rate 403, 408
mechanical damper 414
mechanical damper model 414
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>mechanical mass</td>
<td>415</td>
</tr>
<tr>
<td>mechanical mass model</td>
<td>415</td>
</tr>
<tr>
<td>mechanical modeling</td>
<td>255</td>
</tr>
<tr>
<td>mechanical restrainer</td>
<td>416</td>
</tr>
<tr>
<td>mechanical restrainer model</td>
<td>416</td>
</tr>
<tr>
<td>mechanical spring</td>
<td>418</td>
</tr>
<tr>
<td>mechanical spring model</td>
<td>418</td>
</tr>
<tr>
<td>mechanical systems</td>
<td>413</td>
</tr>
<tr>
<td>messages, error</td>
<td>475</td>
</tr>
<tr>
<td>m-factor (multiplicity factor) accessing inherited</td>
<td>191</td>
</tr>
<tr>
<td>attributes for example of using</td>
<td>191</td>
</tr>
<tr>
<td>using</td>
<td>190</td>
</tr>
<tr>
<td>minimum (min) function</td>
<td>106</td>
</tr>
<tr>
<td>mixed conservative and signal-flow systems</td>
<td>27</td>
</tr>
<tr>
<td>mixed-signal components</td>
<td>420</td>
</tr>
<tr>
<td>mixer</td>
<td>454</td>
</tr>
<tr>
<td>mixer model</td>
<td>453, 454</td>
</tr>
<tr>
<td>model file</td>
<td>240</td>
</tr>
<tr>
<td>modeling models</td>
<td>245</td>
</tr>
<tr>
<td>in modules</td>
<td>239</td>
</tr>
<tr>
<td>library of samples</td>
<td>291</td>
</tr>
<tr>
<td>using in a Verilog-A module</td>
<td>239</td>
</tr>
<tr>
<td>using with a Verilog-A</td>
<td>239</td>
</tr>
<tr>
<td>modulator model</td>
<td>8-bit PCM 457</td>
</tr>
<tr>
<td>AM</td>
<td>441</td>
</tr>
<tr>
<td>FM</td>
<td>452</td>
</tr>
<tr>
<td>PM</td>
<td>461</td>
</tr>
<tr>
<td>QPSK</td>
<td>466</td>
</tr>
<tr>
<td>quadrature amplitude 16-ary</td>
<td>464</td>
</tr>
<tr>
<td>module keyword</td>
<td>32</td>
</tr>
<tr>
<td>modules</td>
<td></td>
</tr>
<tr>
<td>analog behavior of defining</td>
<td>37</td>
</tr>
<tr>
<td>behavioral description</td>
<td>37</td>
</tr>
<tr>
<td>capacitor example</td>
<td>40</td>
</tr>
<tr>
<td>child, definition</td>
<td>504</td>
</tr>
<tr>
<td>declaring</td>
<td>32</td>
</tr>
<tr>
<td>definition</td>
<td>32, 505</td>
</tr>
<tr>
<td>format</td>
<td>32</td>
</tr>
<tr>
<td>format example</td>
<td>32</td>
</tr>
<tr>
<td>hierarchy of</td>
<td>181</td>
</tr>
<tr>
<td>instantiating in other modules</td>
<td>182</td>
</tr>
<tr>
<td>instantiation statement, example</td>
<td>183, 184</td>
</tr>
<tr>
<td>interface declarations</td>
<td>33</td>
</tr>
<tr>
<td>interface, declaring</td>
<td>33</td>
</tr>
<tr>
<td>internal nodes</td>
<td>41</td>
</tr>
<tr>
<td>name</td>
<td>34</td>
</tr>
<tr>
<td>netlist instantiation of</td>
<td>43</td>
</tr>
<tr>
<td>using nodes</td>
<td>71</td>
</tr>
<tr>
<td>non-Verilog-A overview</td>
<td>189</td>
</tr>
<tr>
<td>overview</td>
<td>32</td>
</tr>
<tr>
<td>RLC circuit example</td>
<td>42</td>
</tr>
<tr>
<td>top-level</td>
<td>181</td>
</tr>
<tr>
<td>transformer example</td>
<td>182</td>
</tr>
<tr>
<td>voltage deadband amplifier example</td>
<td>39</td>
</tr>
<tr>
<td>MOS thin-film transistor</td>
<td>435</td>
</tr>
<tr>
<td>MOS thin-film transistor model</td>
<td>435</td>
</tr>
<tr>
<td>MOS transistor (level 1)</td>
<td>433</td>
</tr>
<tr>
<td>MOS transistor model (level 1)</td>
<td>433</td>
</tr>
<tr>
<td>motor model</td>
<td></td>
</tr>
<tr>
<td>behavioral description</td>
<td>248</td>
</tr>
<tr>
<td>DC</td>
<td>346</td>
</tr>
<tr>
<td>three-phase</td>
<td>348</td>
</tr>
<tr>
<td>multilevel hierarchical designs</td>
<td>234</td>
</tr>
<tr>
<td>multiple cellviews, using for instances</td>
<td>216</td>
</tr>
<tr>
<td>multiplexer model</td>
<td>363</td>
</tr>
<tr>
<td>multiplier model</td>
<td>383</td>
</tr>
</tbody>
</table>

**N**

N 506

N JFET transistor model 436

named branches 72

names, escaped 47

naming requirements for SPICE-mode netlisting 483

NAND Gate 327

NAND gate model 327

natural log function model 384

natural logarithm function 106

natures 63

access function for 64

attributes 64

base, declaring 64

base, definition 63

binding with potential and flow 66

declaring 63

definition 506

deriving from other natures 63

requirements for 63

net disciplines 70

description attribute for 70

netlisting Verilog-A modules 235

netlists creating 480
example using cellviews 228
including files in 481
including Verilog-A modules in 235
instantiating module description files in 43
n-channel TFT device 254
preparing to display waveforms 240
typographic conventions used for 21
VCO2 example 237
New Library form 203
new-line characters as white space 46
displaying 166
Newton-Raphson method definition 506
used to evaluate systems 271
no_rigid_switch_branch attribute 279
nodal analysis 271
node data type 70
nodes 26
assumed to be infinitely small 270
connecting instances with 184
declaring 70
definition 506
matching sizes required when connected 185
as module ports 71
reference, definition 507
reference, potential of 27
scalar 70
values associated with 27
vector, declaring 70
vector, definition 70
ways of using 71
noise functions
flicker_noise 131
noise_table 131
noise source model 455
noise_table function 131
noise_table simulator function 131
nonlinearity, announcing and handling 121
NOR Gate 330
NOR gate model 330
normal (gaussian) distribution function 134
normal distribution function 134
NOT Gate 329
NOT gate model 329
NPN bipolar junction transistor model 437
NR method, definition 506
NULL statements 497
numbers 48
numerator-denominator Laplace transforms 159
numerator-denominator Z-transforms 164
numerator-pole Laplace transforms 158
numerator-pole Z-transforms 163
O
o or O format character 167
offset measurement 403
offset measurement model 403
one-line comment 46
opamp model 298, 354
open circuit fault 297
open circuit fault model 297
Open Configuration or Top Cellview form 219
opening
design 225
file 171
opening a configuration and associated schematic 219
operation 244
operational amplifier model 298
operators 87 to 94
analog 143
association of 88
binary 90
bitwise 93
circular integrator 147
delay 150
idtmod 147
precedence 95
precedence of 88, 95
string 95
ternary 94
time derivative 144
time integral 145
unary 89
or (event OR) 92
OR Gate 328
OR gate model 328
OR operator, event 110
order of evaluation, changing 88
ordered lists, mapping nodes with 183
ordinary identifiers 47
oscillator model
digital voltage controlled 450
voltage-controlled 469
overriding parameter values 185
by name 186
from the instantiation statement 185
in instances 185
overview
  analog events 109
  modules 32
  operators 88
  system simulation 24
overview of probes and sources 274

P
P 506
P_C constant 289
P_CELSIUS0 constant 289
P_EPS0 constant 289
P_H constant 289
P_K constant 289
P_Q constant 289
P_U0 constant 289
parallel register model, 8-bit 344
parallel register, 8-bit 344
parameters 36, 54
  aliases 59
  array values as 188
  arrays of 58
  attributes for 55
  changing during compilation 55
  changing of cellview bound with an instance 219
  changing of cellview not currently bound with an instance 220
  changing value of when bound with an instance 219
  changing value of when not bound with an instance 220
  must be constants 54
  declaration, definition 506
  declaring 54
  default value required 56
  defaults, overriding with Edit Object Properties form 218
  definition 506
  deleting from cellviews 221
  dependence on other parameters 55
  enumerated values as 189
  examining current values of 226
  inherited 55
  names 36
not displayed in Edit Object Properties form unless overridden 227
overrides
detecting 125
overriding values with module instantiation statement 185
permissible values for, specifying 56
specified in modules, modifying 218
string 59
string values as 189
type specifier optional 56
type, specifying 56
parentheses
  changing evaluation order with 88
  using to exclude end point in range 57
parsing, errors during 208
paths
  absolute 479
  relative 479
  specifying with CDS_VLOGA_INCLUDE environment variable 480
PCM demodulator model, 8-bit 456
PCM demodulator, 8-bit 456
PCM modulator model, 8-bit 457
PCM modulator, 8-bit 457
period of signal, example of calculating 123
permissible values for parameters, specifying 56
permissible values, specifying 56
phase detector model 458
phase locked loop model 459
digital 449
pin direction 205
pins
  adding to blocks 206
  deleting 206
  direction of, in symbols 205
  specifying information for 212
  specifying name seed for 205
PLL model 459
digital 449
PLL, definition 451
plotting variables 240
PM demodulator 460
PM demodulator model 460
PM modulator 461
PM modulator model 461
Poisson distribution 136
Poisson distribution function 136
polynomial 385
polynomial model 385
port branches 275
  contrasted with simple port 275
  monitoring flow with 275
port bus, defining 71
port connection rules 185
port declaration example 36
port direction 35
port type 35
ports 34
  bidirectional 35
  declaring 34
  defining by listing nodes 71
  direction, declaring 35
  instance, mapping to defining module 183
  names, using to connect instances 184
  type of, declaring 35
  undeclared types as 35
potential definition 506
  in electrical systems 27
  probes 274
  sources, definition 275
  sources, equivalent circuit model 276
  sources, switching to flow sources 277
potential law. See Kirchhoff’s Laws 27
power (pow) function 106
power consumption, specifying 170
power electronics components 429
power function model 386
power meter model 404
power sink model, constant 299
precedence of operators 88, 95
preparing a library 202
primitives definition 506
  instantiating in Verilog-A modules 188
probe model delta 398
  find event 399
  signal statistics 399, 401, 409
probes 274
definition 274, 507
flow 274
potential 274
  reasons for using 274
procedural assignment statement 76
procedural assignment statements in the analog block 76
procedural control constructs 75
proportional controller model 322
proportional derivative controller 323
proportional derivative controller model 323
proportional integral controller model 324
proportional integral derivative controller model 325
pump model, charge 445

Q
Q (charge) meter model 406
QAM 16-ary demodulator model 462
QPSK demodulator model 465
QPSK modulator model 462, 466
QPSK, definition 462
quadrature amplitude 16-ary modulator model 464
quadrature phase shift key demodulator model 465
quadrature phase shift key modulator model 466
quantities defining 214
  parameters for 215
quantity statement
  modifying absolute tolerances with 485
  syntax 214
quantity.spectre file
  overriding values in 215
  specifying quantities with 214
quantizer model 364
querying the simulation environment 123

R
R 507
random bit stream generator model 467
random numbers, generating 132
$random simulator function 132
range
  for integer numbers 52
  for real numbers 53
rate of change, controlling with slew filter 154
reading from a file 174
real argument not supported as direction
argument 499
real constants 49
scale factors for 49
syntax 48
real numbers 48, 52
attributes for 53
converting to integers 53
declaring 52
range permitted 53
reciprocal model 387
rectangular hysteresis model 358
rectifiers
behavioral description for 246
example 243
reference directions 27
associated 27
definition 507
illustrated 27
reference directions, choosing 264
reference nodes 27
compatibility of 185
definition 507
potential of 27
related documents 19
relative paths 479
relative tolerance 272
relay
example 120
model, electromagnetic 347
reltol (relative tolerance) 272
repeat loop statement 81
repeat statement 81
repeater 365
repeater model 365
`resetall compiler directive 197
resetting directives to default values 197
resistor model 310
self-tuning 303
untrimmed 307
restrainer model 416
restrictions on using analog operators 144
rigid branches, attribute for 279
rise times, setting default for 197
RLC Circuit 281
RLC circuit 41, 42
RLC circuit model 281
rms, definition 395
road model 258, 417
RS-Type Flip-Flop 338
RS-type flip-flop model 338
rules, for connecting instances 185
run time binding, definition 507
S
S 507
s or S format character 168
sample-and-hold amplifier model (ideal) 426
sampler model 407
saturating integrator model 366
saveahdlvars option 240
saving Verilog-A variables 240
scalar node 70
scale factors, for real constants 49
schematic cellviews
instantiating in Verilog-A
components 238
opening 219
opening in Cadence analog design environment 225
rules for instantiating in Verilog-A
modules 238
schHiCreateBlockInst SKILL function 206
Schottky Diode 439
Schottky diode model 439
scope
definition 507
named block defines new 79
of discipline identifiers 66
rules 47
self-tuning resistor 303
self-tuning resistor model 303
semiconductor components 432
sequential block statement 79
serial register model, 8-bit 345
serial register, 8-bit 345
shdl_strchr function 97
shdl_strchr string function 102
shdl_strcspn function 97
shdl_strcspn string function 102
shdl_strings.vams file
location in hierarchy 96
used to implement string functions 96
shdl_strchr function 97
shdl_strchr string function 103
shdl_strcspn function 97
shdl_strcspn string function 103
shdl_strstr function 97
shdl_strstr string function 104
shifter model, level 337, 360
shock absorber model 257
short circuit fault 300
short circuit fault model 300
short circuiting, of expressions 95
Show Instance Table button 231
sigma-delta converter (first-order) 425
sigma-delta converter model (first order) 425
signal driver model, differential 355
signal statistics probe 409
signal statistics probe model 399, 401, 409
signal values
  modifying with branch contribution statement 76
  obtaining and setting 126
signal values, obtaining and setting 126
signal-flow discipline 67
signal-flow systems 27
  modeling supported by Verilog-A 27
signal-flow disciplines used to define 72
signed number 388
signed number model 388
signs, requesting in output 167
simple filename 480
simple implicit diode 281
simple implicit diode model 281
simulating a system 271
simulation
  overview 24
  preparing for 477
Simulation Environment Options form 238
simulation environment, querying 123
simulation time, obtaining current 123
simulation view lists 238
simulator flow 28
simulator functions
  $dist_chi_square 136
  $dist_erlang 138
  $dist_exponential 135
  $dist_normal 134
  $dist_poisson 136
  $dist_t 137
  $dist_uniform 133
  $random 132
  ac_stim 130
  analysis 128
  bound_step 121
  discontinuity 119
  flicker_noise 131
  last_crossing 122
  limiting function 121
  noise_table 131
  white_noise 130
  sin function 107
  sine function 107
  single shot model 427
  sinh function 107
  sink model, constant power 299
  sinusoidal source
    swept, model 367
    variable frequency, model 370
  sinusoidal stimulus, implementing with
    ac_stim 130
  sinusoidal waveforms, controlling with slew filter 155
  sizes, of connected terminals and nodes 185
  SKILL function 206
SKILL functions,
  schHiCreateBlockInst 206
  slew filter 154
  slew rate measurement model 403, 408
  small-signal AC sources 130
  small-signal noise sources 130
  smoothing piecewise constant waveforms 151
  soft current clamp model 301
  soft voltage clamp model 302
  source model
    audio 443
    noise 455
    swept sinusoidal 367
    three-phase 368
    variable frequency sinusoidal 370
  sources 275
    controlled 276
    current-controlled current 277
    current-controlled voltage 277
    definition 274, 507
    flow 275
    linear conductor model 280
    linear resistor model 281
    potential 275
    reasons for using 274
    RLC circuit model 281
    simple implicit diode model 281
    unassigned 277
    voltage-controlled current 276
    voltage-controlled voltage 276
  space
    displaying or printing 167
<table>
<thead>
<tr>
<th>White</th>
<th>Special Characters</th>
<th>Special Characters, Displaying</th>
<th>Spectre</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Netlist File</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Creating 480</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Primitives, Instantiating In Verilog-A Modules 188</td>
</tr>
<tr>
<td>Spectre/Spectreverilog Interface (Spectre Direct) 215</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpectreHDL Equivalent String Functions 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Table Of 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpectreHDL String Functions, Verilog-A Equivalents For 98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpectreHDL String Functions, Verilog-A Replacements For 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpectreVerilog 201</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPICE-Mode Netlisting, Naming Requirements For 483</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spring Model 256, 418</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sqr Function 106</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Brackets, Meaning Of, In Syntax 21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Model 389</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Root Function 106</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Square Root Model 390</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$sscanf 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$sscanf Function 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$sscanf Operator</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Details 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Constants 288</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Mathematical Functions 106</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Copy Operator 99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Copy Operator, Details 99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Functions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$sscanf 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atol 100</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atoreal 101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation 99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy 99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Getc 101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Len 101</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strchr 102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strcspn 102</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strchr 103</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strspn 103</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strstr 104</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SpectreHDL Equivalent 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verilog-A 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>String Parameters, Declaring 59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strings</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atol Operator 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atoreal Function 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Characters In Specified Set 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparison Operators 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Comparison Operators For 99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Concatenation Operator 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Converting To Integer 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Converting To Real 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy Operators 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>First Position Of Subset 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Getc Function 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Last Location Of Character In 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Len Function 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Location Of Character In 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number Of Characters In 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operators And Functions 95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strchr Function 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strcspn 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strings.vams File And String Functions 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strchr Function 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strspn 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shdl_strstr Function 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Size Of Character Set In 97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substr Function 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substrings Off 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verilog-A Equivalent Functions For 98</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strings, As Parameter Values 189</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$Strobe</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Description 166, 170</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example 168</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structural Definitions, Definition 507</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Structural Descriptions, Undeclared Port Types In 35</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Student's T Distribution 137</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Student's T Distribution Function 137</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Substr Function 96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtractor Model 391</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Four Numbers 392</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Full 343</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Half 342</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Suggestions For Updating Models 496</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Svcvs Primitive 188</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Swept Sinusoidal Source Model 367</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switch 317</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branch, Creating 78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches 78, 277, 278</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Branches, Value Retention For 278</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Model 317</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Switch View List</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
two-phase transformer model 375
type specifier, optional on parameter
declaration 56
typographic and syntax conventions 20

U
unary operators 89
defined 89
precedence of 89
unary reduction operators 89
unassigned sources 277
‘undef compiler directive 196
undefining text macros 196
underscore, in identifiers 47
uniform distribution 133
uniform distribution function 133
unit attribute
description 64
for integers 52, 191, 279
for parameters 55
for reals 53
requirements for 65
units (scale factors) for real numbers 49
untrimmed
capacitor model 305
inductor model 306
resistor model 307
user-defined functions 177
calling 179
declaring 177
declaring analog 177
restrictions on 178

V
V 508
.va file extension 478
value retention for switch branches 278
value-to-flow converter model 369
variable frequency sinusoidal source
model 370
variable-gain amplifier model, voltage-controlled 309
variable-gain differential amplifier
model 371
variables
displaying waveforms of 240
variables, choosing 264
VCO model 469
VCO, definition 449
vector nodes, definition 70
vectors, arguments represented as 156
Verilog and VHDL 239
Verilog, digital
cannot instantiate below Verilog-A
module 239
wiring to Verilog-A components 239
Verilog-A
definition 508
language overview 24
.va extension for files 478
Verilog-A cellviews
creating 210
creating from a symbol or block 206
creating from existing Verilog-A
cellviews 217
creating from scratch 210
creating from symbols or blocks 206
editing outside of the analog design
environment 208
veriloga cellviews, creating with VerilogA-Editor 210
Verilog-A module description, creating 478
Verilog-A string functions, table of 96
vertical bars, meaning of, in syntax 20
VHDL
cannot instantiate below Verilog-A
module. 239
wiring to Verilog-A components 239
voltage clamp model
hard 296
soft 302
voltage deadband amplifier 38
model 308
voltage meter model 411
voltage source model
current-controlled 314
voltage-controlled 313
voltage-controlled current source 276
voltage-controlled current source
model 315
voltage-controlled oscillator
model 469
model, digital 450
voltage-controlled variable-gain amplifier
model 309
voltage-controlled voltage source 276
voltage-controlled voltage source
model 313
W

waveforms, displaying  240
wheel  419
wheel model  260, 419
while loop statement  82
while statement  82
white space  46
white_noise function  130
white_noise simulator function  130
winding model, magnetic  374
wire (predefined empty discipline)  67
writing to a file  174

X

XNOR Gate  332
XNOR gate model  332
XOR Gate  331
XOR gate model  331

Z

Z (impedance) meter  412
Z (impedance) meter model  412
zero crosses, detecting  112
zero-denominator Laplace transforms  157
zero-denominator Z-transforms  163
zero-pole Laplace transforms  157
zero-pole Z-transforms  162
zi_nd Z-transform filter  164
zi_np Z-transform filter  163
zi_zd Z-transform filter  163
zi_zp Z-transform filter  162
Z-transform filters  161
Z-transforms
  introduction  161
    numerator-denominator form  164
    numerator-pole form  163
    zero-denominator form  163
    zero-pole form  162