Notice

The information contained in this document is subject to change without notice.

Agilent Technologies makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Agilent Technologies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms that apply to this software product is available upon request from your Agilent Technologies representative.

Restricted Rights Legend

Use, duplication or disclosure by the U. S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DoD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Agilent Technologies
395 Page Mill Road
Palo Alto, CA 94304 U.S.A.


Acknowledgments

Mentor Graphics is a trademark of Mentor Graphics Corporation in the U.S. and other countries.

Microsoft®, Windows®, MS Windows®, Windows NT®, and MS-DOS® are U.S. registered trademarks of Microsoft Corporation.

Pentium® is a U.S. registered trademark of Intel Corporation.

PostScript® and Acrobat® are trademarks of Adobe Systems Incorporated.

UNIX® is a registered trademark of the Open Group.

Java™ is a U.S. trademark of Sun Microsystems, Inc.
Contents

1 Overview and Benefits
   Analog Modeling ................................................................. 1-1
   Hardware Description Languages ........................................ 1-1
   Verilog-A ............................................................................. 1-2
   Compact Models ............................................................... 1-2
   Simulation ......................................................................... 1-4

2 Verilog-A Modules
   Declaring Modules .......................................................... 2-1
   Module Instantiation ......................................................... 2-1
   Ports .................................................................................. 2-2
   Describing Analog Behavior ............................................. 2-2
   Branches ............................................................................ 2-3
   Analog Signals ................................................................. 2-3
      Accessing Net and Branch Signals ................................... 2-3
   User-defined Analog Functions ....................................... 2-4
   Hierarchical Structures .................................................... 2-6
      Module Instance Parameter Value Assignment ............... 2-6
      Port Assignment ......................................................... 2-7
      Scope ........................................................................... 2-8

3 Lexical Conventions
   White Space ...................................................................... 3-2
   Comments ......................................................................... 3-2
   Operators .......................................................................... 3-2
   Strings .............................................................................. 3-2
   Numbers ........................................................................... 3-3
      Integer Numbers .......................................................... 3-3
      Real Numbers ................................................................ 3-3
      Scale Factors .............................................................. 3-3
   Keywords .......................................................................... 3-4
   Identifiers ......................................................................... 3-4
   System Tasks and Functions ............................................ 3-5
   Compiler Directives ......................................................... 3-5

4 Data Types
   Integer ............................................................................. 4-1
   Real ................................................................................ 4-1
   Type Conversion ........................................................... 4-1
   Net Discipline ............................................................... 4-2
   Ground Declaration ....................................................... 4-3
Absolute Delay Operator ................................................................. 7-3
Transition Filter .............................................................................. 7-4
Slew Filter ....................................................................................... 7-6
Last Crossing Function ................................................................. 7-7
Limited Exponential ....................................................................... 7-7
Laplace Transform Filters ............................................................... 7-8
  laplace_zp() ............................................................................... 7-8
  laplace_zd() ............................................................................... 7-9
  laplace_np() ............................................................................. 7-10
  laplace_nd() ............................................................................. 7-10
Z-Transform Filters ........................................................................ 7-11
  zi_zp() .................................................................................... 7-11
  zi_zd() .................................................................................... 7-12
  zi_np() .................................................................................... 7-13
  zi_nd() .................................................................................... 7-14

8 Analog Events
  Global Events .................................................................................. 8-1
    The initial_step Event ................................................................. 8-1
    The final_step Event ................................................................. 8-2
  Global Event Return Codes ............................................................ 8-2
  Monitored Events .......................................................................... 8-3
    The cross Function .................................................................... 8-3
    The timer Function .................................................................... 8-4
  Event or Operator .......................................................................... 8-5
  Event Triggered Statements ............................................................ 8-5

9 Verilog-A and the Simulator
  Environment Parameter Functions .................................................. 9-1
    The temperature Function .......................................................... 9-1
    The abstime Function ................................................................. 9-1
    The realtime Function ............................................................... 9-1
    The Thermal Voltage Function .................................................... 9-2
  Controlling Simulator Actions ....................................................... 9-2
    Bounding the Time Step ............................................................. 9-2
    Announcing Discontinuities ......................................................... 9-3
  Analysis Dependent Functions ....................................................... 9-3
    Analysis Function ....................................................................... 9-3
    AC Stimulus Function ................................................................ 9-5
    Noise Functions .......................................................................... 9-5

10 System Tasks and I/O Functions
  File Input/Output Operations ........................................................ 10-1
Chapter 1: Overview and Benefits

This chapter introduces the Verilog-A language and software in terms of its capabilities, benefits, and typical use.

Analog Modeling

Analog modeling enables designers to capture high-level behavioral descriptions of components in a precise set of mathematical terms. The analog module's relation of input to output can be related by the external parameter description and the mathematical relations between the input and output ports.

Analog models give the designer control over the level of abstraction with which to describe the action of the component. This can provide higher levels of complexity to be simulated, allow faster simulation execution speeds, or can hide intellectual property.

An analog model should ideally model the characteristics of the behavior as accurately as possible, with the trade off being model complexity, which is usually manifested by reduced execution speed. For electrical models, besides the port relationship of charges and currents, the developer may need to take thermal behavior, physical layout considerations, environment (substrate, wires) interaction, noise, and light, among other things into consideration. Users prefer that the model be coupled to measurable quantities. This provides reassurance in validating the model, but also provides a means to predict future performance as the component is modified.

Models often have to work with controlling programs besides the traditional simulator. Optimization, statistical, reliability, and synthesis programs may require other information than which the model developer was expecting.

Hardware Description Languages

Hardware description languages (HDLs) were developed as a means to provide varying levels of abstraction to designers. Integrated circuits are too complex for an engineer to create by specifying the individual transistors and wires. HDLs allow the performance to be described at a high level and simulation synthesis programs can then take the language and generate the gate level description.
Overview and Benefits

Verilog and VHDL are the two dominant languages; this manual is concerned with the Verilog language.

As behavior beyond the digital performance was added, a mixed-signal language was created to manage the interaction between digital and analog signals. A subset of this, Verilog-A, was defined. Verilog-A describes analog behavior only; however, it has functionality to interface to some digital behavior.

**Verilog-A**

Verilog-A provides a high-level language to describe the analog behavior of conservative systems. The disciplines and natures of the Verilog-A language enable designers to reflect the potential and flow descriptions of electrical, mechanical, thermal, and other systems.

Verilog-A is a procedural language, with constructs similar to C and other languages. It provides simple constructs to describe the model behavior to the simulator program. The model effectively de-couples the description of the model from the simulator.

The model creator provides the constitutive relationship of the inputs and outputs, the parameter names and ranges, while the Verilog-A compiler handles the necessary interactions between the model and the simulator. While the language does allow some knowledge of the simulator, most model descriptions should not need to know anything about the type of analysis being run.

**Compact Models**

Compact models are the set of mathematical equations that describe the performance of a device. Commercial simulators use compact models to describe the performance of semiconductor devices, most typically transistors.

There is a wide range of modeling categories, including neural nets, empirical, physical, and table based. Each has distinct advantages and disadvantages as listed in Table 1-1 below.
For electrical modeling, most compact device models use empirical modeling based on physical models. This provides the best combination of execution speed, accuracy, and prediction. However, non-physical behavior may result when the equations are used outside their fitting range. Model creators should also be aware of the issues around parameter extraction. If a model’s parameters cannot easily or accurately be extracted, the model will not be successful.

Once created, a compact model can be implemented in a simulator in a variety of methods (see Table 1-2). Each method has its own advantages and disadvantages, but in general, the simpler the interface, the less capable it is.

Table 1-1. Modeling Categories

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical</td>
<td>Predicts performance best</td>
<td>Must understand physics</td>
</tr>
<tr>
<td></td>
<td>Extrapolates</td>
<td>Slow</td>
</tr>
<tr>
<td>Empirical</td>
<td>Reasonably good prediction</td>
<td>Can give non-physical behavior</td>
</tr>
<tr>
<td></td>
<td>Fast</td>
<td></td>
</tr>
<tr>
<td>Tabular</td>
<td>Very general</td>
<td>Cannot extrapolate</td>
</tr>
<tr>
<td></td>
<td>Easy to extract</td>
<td>Minimal parameter info</td>
</tr>
<tr>
<td>Neural net</td>
<td>Very general</td>
<td>Cannot extrapolate</td>
</tr>
<tr>
<td></td>
<td>Reasonable execution speed</td>
<td>Minimal parameter info</td>
</tr>
</tbody>
</table>

Table 1-2. Compact Model Simulator Implementation Methods

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macro model</td>
<td>Simple, portable</td>
<td>Limited to available primitives</td>
</tr>
<tr>
<td>Proprietary interface</td>
<td>Power, fast</td>
<td>Need access to simulator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not portable</td>
</tr>
</tbody>
</table>
Overview and Benefits

<table>
<thead>
<tr>
<th>Type</th>
<th>Advantage</th>
<th>Disadvantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public interface</td>
<td>Reasonably powerful</td>
<td>Usually missing some capability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Not portable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unique complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Slow</td>
</tr>
<tr>
<td>AHDLs (Verilog-A)</td>
<td>Simple</td>
<td>Language has some restrictions</td>
</tr>
<tr>
<td></td>
<td>Power</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Portable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Protected</td>
<td></td>
</tr>
</tbody>
</table>

The most powerful interface is the proprietary interface to the simulator. For many reasons, most typically intellectual property protection, the proprietary interface is not made public. This is because the interface usually requires such intimate details of the simulator analysis operation that a clever investigator could discern much detail about the inner workings of the simulation algorithms.

On the other hand, a detailed, complex interface also requires a detailed understanding by the model developer to properly access the functionality of the analysis. This can require as much effort as the development of the model itself. If the model is to be added using this interface to other simulators, often the effort of learning one interface does not provide much advantage in learning the nuances of the other.

Simulator vendors often provide simplified interfaces, either a scaled-back code level interface, or a custom symbolic interface. The simplicity always comes at a price of reduced functionality, decreased execution speed, or a lack of portability.

Analog Hardware Description languages (AHDLs) solve most of these problems, except the problem of execution speed. However, compiler technology in Advanced Design System provides the abstraction and simplicity of Verilog-A with an execution speed with a factor of two of code level interfaces.

Simulation

Most analog simulators evolved from the SPICE program released by UC Berkeley. The analog simulator solves a simple set of relations for a large number of unknowns to provide the designer with the voltage and currents at each of the nodes in the circuit as either a function of time or frequency. The electrical relations are simply the Kirchoff current law and voltage loop laws:
• The instantaneous currents from all branches entering a node must sum to zero.
• The instantaneous voltages around any closed loop must sum to zero.

Figure 1-1. Kirchoff’s Current and Voltage Laws.

The program solves these equations using an algorithm known as Newton-Raphson. The program guesses a solution to the relation:

\[ F(v, t) = 0 \]

It next calculates if the solution is close enough, and if it is, it stops. If the program needs to find the next guess, it calculates the Jacobian of the function (the set of partial derivatives), which provides a pointer towards the real solution. This is best seen in a simple case where \( F(v, t) \) is one dimensional:
Overview and Benefits

The program calculates \( F(x_1) \) and then uses the derivative (the slope of the line) to estimate the zero-crossing. It then uses this new point, \( x_2 \), as the next guess. As long as the function is continuous and smooth (and the program does not get stuck in local minima), the program will quickly approach, or converge to, the actual point where \( F(X) \) crosses zero.

In modern simulation programs the trick is to quickly find the solution for a large number of variables, and for variables that may have strong (nonlinear) relations to independent variables.

During a simulation, the program queries each element in the circuit for information. The resistor, the capacitor, or the transistor, for example, needs to report back to the simulator its behavior at a particular guess. It must also report back its slope with respect to the voltages at that point.

The behavior of each element is described by mathematical equations that, taken as a group, are called a compact device model. The compact device model can be very simple or very complicated. For example, in the simplest case, a resistor can be described by Ohm's law:

\[
I = \frac{V}{R}
\]

The derivative is a constant, the inverse of the resistance.

However, even for simple components like resistors, the models can become rather complicated when other effects are added, such as, self-heating, self-inductance, thermal noise, etc. The Verilog-A compiler manages all of the necessary interfaces so that, for the most part, the developer need only be concerned with model behavior.
Chapter 2: Verilog-A Modules

This chapter discusses the concept of Verilog-A modules. The chapter shows the basic structure of a module declaration, how to define parameters and ports, and how to define a simple analog block.

Declaring Modules

The module declaration provides the simulator with the name of the module, the input and output ports, parameter information, and the behavioral description of the model. Top-level modules are modules which are included in the source text but are not instantiated. Module definitions cannot contain the text of another module definition. A module definition can nest another module by instantiating it. For more information, refer to “Hierarchical Structures” on page 2-6.

Module Instantiation

Syntax

module | macromodule module_identifier [(port {, port, ...})]
// module statements
endmodule

where module_identifier is the name of the module and the optional list of port name(s) defines the connections to the module.

Example

The simplest example is a resistor.

```
'include "disciplines.vams"
module R(p,n);
    electrical p,n;
    parameter real R=50.0;
    analog
    V(p,n) <+ R * I(p,n);
endmodule
```

The first line provides common definitions. The line `module R(p, n);` declares the module name to be R and that it has 2 ports, named p and n, which the next line further describes by attributing the electrical discipline to the ports.
Verilog-A Modules

This module has one parameter, \( R \), which is declared as a real type with a default value of 50.0. Parameters provide a way to pass information into the module at the time of instantiation.

The analog block, in this example a single line, describes the behavior using a voltage contribution statement to assign the voltage based on the access function value of \( I() \) times \( R \).

Ports

Ports provide a way to connect modules to other modules and devices. A port has a direction: input, output, or inout, which must be declared. The ports are listed after the module declaration. The port type and port direction must then be declared in the body of the module.

Examples

```verilog
module resistor(p,n);
inout p,n;
electrical p,n;
```

```verilog
module modName(outPort, inPort);
output outPort;
input inPort;
electrical outPort, inPort;
```

Ports can support vectors (buses) as well.

Describing Analog Behavior

The analog behavior of the component is described with procedural statements defined within an analog block. During simulation, all of the analog blocks are evaluated. Each module is evaluated in the design as though it were contributing concurrently to the analysis.

Syntax

```verilog
analog block_statement
```
Examples

```
analog V(n1, n2) <+ 1; // A simple 1 volt source
analog begin // A multi-statement analog block
  vin = V(in);
  if (vin >= signal_in_dead_high)
    vout = vin - signal_in_dead_high;
  else
    if (vin <= signal_in_dead_low)
      vout = vin - signal_in_dead_low;
    else
      vout = 0.0;
  V(out) <+ vout;
end
```

Branches

A branch is defined as a path between two nets. A branch is conservative if both nets are conservative and two associated quantities, potential and flow, are defined for the branch. If either net is a signal-flow net, then the branch is defined as a signal-flow branch with either a potential or flow defined for the branch.

Syntax

```
branch list_of_branches
```

where list_of_branches is a comma-separated list of branch names.

Analog Signals

Analog signals are signals associated with a discipline that has a continuous domain. Their value can be accessed and set via various functions and contribution statements. This section describes the analog signal functions. It describes how to access signal data from nodes and vectors, as well as how to use the contribution operator.

Accessing Net and Branch Signals

Signals on nets and branches can be accessed only by the access functions of the associated discipline. The name of the net or the branch is specified as the argument to the access function.
Examples

\[
\text{Vin} = V(\text{in}); \\
\text{CurrentThruBranch} = I(\text{myBranch}); \\
\]

Indirect branch assignment

An indirect branch assignment is useful when it is difficult to solve an equation. It has this format,

\[
V(n) : V(p) == 0; \\
\]

which can be read as “find \(V(n)\) such that \(V(p)\) is equal to zero.” This example shows that node \(n\) should be driven with a voltage source and the voltage should be such that the given equation is satisfied. \(V(p)\) is probed and not driven.

**Note** Indirect branch assignments are allowed only within the analog block.

Branch Contribution Statement

A branch contribution statement typically consists of a left-hand side and a right-hand side, separated by a branch contribution operator. The right-hand side can be any expression which evaluates to (or can be promoted to) a real value. The left-hand side specifies the source branch signal to assign the right-hand side. It consists of a signal access function applied to a branch. The form is,

\[
V(n1, n2) <+ expression; \\
I(n1, n2) <+ expression; \\
\]

Branch contribution statements implicitly define source branch relations. The branch extends from the first net of the access function to the second net. If the second net is not specified in the call, the global reference node (ground) is used as the reference net.

User-defined Analog Functions

Analog functions provide a modular way for a user-defined function to accept parameters and return a value. The functions are defined as analog or digital and must be defined within modules blocks.

The analog function is of the form:
analog function {real|integer} function_name;
  input_declaration;
  statement_block;
endfunction

The input_declaration describes the input parameters to the function as well as any variables used in the statement block:

  input passed_parameters;
  real parameter_list;

The statement_block and analog function:

  • can use any statements available for conditional execution
  • cannot use access functions
  • cannot use contribution statements or event control statements
  • must have at least one input declared; the block item declaration declares the type of the inputs as well as local variables used
  • cannot use named blocks
  • can only reference locally-defined variables or passed variable arguments

The analog function implicitly declares a variable of the same name as the function, function_name. This variable must be assigned in the statement block; its last assigned value is passed back.

Example

  analog function real B_of_T;
    input B, T, T_NOM, XTB;
    real B, T, T_NOM, XTB;
    begin
      B_of_T = B * pow(T / T_NOM, XTB);
    end
  endfunction

The function is called by the line,

  BF_T = B_of_T(BF, T, T_NOM, XTB);
Hierarchical Structures

Verilog-A supports hierarchical descriptions, whereby modules can instantiate other modules. This section describes the procedure for implementing and calling hierarchical models.

Syntax

```
module_or_primate #( {param1(expr), , param2(expr) } ) instance_name
( {node {, node} });
```

Examples

```
phaseDetector #( .gain(2) ) pd1(lo, rf, if_);

vco #( .gain(loopGain/2), .fc(fc) ) vco1(out, lo);
```

Module Instance Parameter Value Assignment

The default parameter values can be overridden by assigning them via an ordered list or explicitly when instantiating a module.

By Order

In this method, the assignment order in the instance declaration follows the order of the parameter declaration in the module declaration. It is not necessary to assign all of the parameters, but all parameters to the left of a declaration must be defined (that is, parameters to the left of the last declaration can not be skipped).

Example

```
// Voltage Controlled Oscillator
module vco(in, out);
inout in, out;
electrical in, out;
parameter real gain = 1, fc = 1;
analog
   V(out) <+ sin(2*`M_PI*(fc*$realtime() + idt(gain*V(in))));
endmodule
...

// Instantiate a vco module name vco1 connected to out and
// lo with gain = 0.5, fc = 2k
vco #(0.5, 2000.0) vco1(out, lo);
```
By Name

Alternatively, instance parameters can be assigned explicitly by their name, where the name matches the parameter name in the module. In this method, only the parameters that are modified from their default values need to be assigned.

Example

```vhdl
// Voltage Controlled Oscillator
module vco(in, out);
inout in, out;
electrical in, out;
parameter real gain = 1, fc = 1;
analog
  V(out) <+ sin(2*M_PI*(fc*realtime() + idt(gain*V(in))));
endmodule
```

```vhdl
// Instantiate a vco module name vco1 connected to out and lo with
// gain = loopGain/2, fc = fc
vco #(.gain(loopGain/2), .fc(fc) ) vco1(out, lo);
```

Port Assignment

Ports can be assigned either via an ordered list or directly by name.

By Order

To connect ports by an ordered list, the ports in the instance declaration should be listed in the same order as the module port definition.

Example

```vhdl
module sinev(n1,n2);
electrical n1,n2;
parameter real gain = 1.0, freq = 1.0;
analog begin
  V(n2,n1) <+ gain * sin(2*M_PI*freq*$abstime);
  $bound_step(0.05/freq);
end
endmodule
...
Verilog-A Modules

// Instantiate a source1 with in->n1, out->n2
sinev #(.gain(G), .freq(F) ) source1(in, out)

By Name

To connect ports by name, both the name of the port in the module definition and the
name of the port in the instance are listed.

module sinev(n1,n2);
electrical n1,n2;
parameter real gain = 1.0, freq = 1.0;
analog begin
   V(n2,n1) <+ gain * sin(2 * 'M_PI * freq * $abstime);
   $bound_step(0.05/freq);
end
endmodule

// Instantiate a source1 with in->n1, out->n2
// Note the order is reversed for the same connection
// result as the By Order example
sinev #(.gain(G), .freq(F) ) source1(.n2(out), .n1(in))

Scope

Verilog-A supports name spaces for the following elements:

- modules
- tasks
- named blocks
- functions
- analog functions

Within each scope only one identifier can be declared. To reference an identifier
directly, the identifier must be declared locally in the named block, or within a
module, or within a named block that is higher in the same branch of the name
hierarchy that contains the named block. If an identifier is declared locally, it will be
used, otherwise the identifier will be searched upwards until it is found, or until a
module boundary is reached.
Chapter 3: Lexical Conventions

This chapter describes the overall lexical conventions of Verilog-A, and how the language defines and interprets various elements such as white space, strings, numbers, and keywords.

Verilog-A consists of lexical tokens (one or more characters) of the form:

- “White Space” on page 3-2
- “Comments” on page 3-2
- “Operators” on page 3-2
- “Strings” on page 3-2
- “Numbers” on page 3-3
- “Keywords” on page 3-4
- “Identifiers” on page 3-4
- “System Tasks and Functions” on page 3-5
- “Compiler Directives” on page 3-5

The source file is free form where spaces, tabs, and newlines are only token separators and have no other significance. Lines can be extended using the line continuation character / where needed.
Lexical Conventions

White Space
White space consists of spaces, tabs, newlines, and form feeds. They separate tokens, otherwise are ignored.

Comments
There are two ways to include comments:

- A single line comment starts with // and continues to the end of the line.
  
  Example
  // This is a single line comment

- Block statements begin with /* and end with */ and cannot be nested but can include single line comments.
  
  Example
  /* This is a block comment which can include any ASCII character */

Operators
Verilog-A has unary (single) operators, binary (double) operators and the conditional operator. Unary operators appear to the left of the operand, and binary between their operands. The conditional operator separates the three operands with two special characters.

Strings
Strings are sequences of characters enclosed by double quotes and contained on one line.

Example
"This is a string."
Numbers

Constant numbers can be specified as integer or a real constants; complex constants are not allowed. Scale factors can be used for readability on real numbers.

Integer Numbers

Integer constants must be specified as a sequence of the digits 0 through 9 in a decimal format with an optional + or - unary operator at the start. The underscore character can be used at any position except the first character as a means to break up the number for readability.

Examples

12345
-122
867_5309

Real Numbers

Real constants follow the IEEE standard for double precision floating point numbers, IEEE STD-754-1985. They can be specified in decimal notation or scientific notation. If a decimal point is used, the number must have at least one digit on each side of the decimal point (e.g., 0.1 or 17.0 are allowed, .1 or 17. are not). As in the integer case, the underscore character is allowed anywhere but the first character and is ignored.

Examples

3.14
1.23e-9
27E9
876_763_300E10

Scale Factors

Scale factors can be used on floating point numbers, but cannot be used with numbers in scientific format. The scale factor symbol and the number cannot have a space between them.
Lexical Conventions

Table 3-1. Scale Factors

<table>
<thead>
<tr>
<th>Scale Factor Symbol</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>10^{12}</td>
</tr>
<tr>
<td>G</td>
<td>10^{9}</td>
</tr>
<tr>
<td>M</td>
<td>10^{6}</td>
</tr>
<tr>
<td>K or k</td>
<td>10^{3}</td>
</tr>
<tr>
<td>m</td>
<td>10^{-3}</td>
</tr>
<tr>
<td>µ</td>
<td>10^{-6}</td>
</tr>
<tr>
<td>n</td>
<td>10^{-9}</td>
</tr>
<tr>
<td>p</td>
<td>10^{-12}</td>
</tr>
<tr>
<td>f</td>
<td>10^{-15}</td>
</tr>
<tr>
<td>a</td>
<td>10^{-18}</td>
</tr>
</tbody>
</table>

Examples

- 2.5m  2.5e-3  0.025
- 0.11M 1.1e5  110000

Keywords

Keywords are predefined non-escaped identifiers. Keywords define the language constructs. They are defined in lowercase only. Appendix A, Reserved Words in Verilog-A, lists all of the keywords, which includes the Verilog-AMS keywords.

Identifiers

Identifiers give objects unique names for reference and can consist of any sequence of letters, digits, the $ character, and the _ (underscore) character. The first character of an identifier can be a letter or underscore, it cannot be the $ character or a digit. Identifiers are case sensitive.
System Tasks and Functions

User-defined tasks and functions use a $ character to declare a system task or system function. Any valid identifier, including keywords (not already in use in this construct), can be used as system task and system function names. Note that for backward compatibility with earlier versions of Verilog-A, this implementation reserves some task and function names.

Examples

	$temperature;
	$strobe("hello");

Compiler Directives

Compiler directives are indicated using the ` (accent grave) character. For more information, refer to Chapter 11, The Verilog-A Preprocessor, for a discussion of compiler directives.
Lexical Conventions
Chapter 4: Data Types

This section describes the various data types that Verilog-A supports as well as shows the correct format and use model. Verilog-A supports integer, real, parameter, and discipline data types.

Integer

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

Example

```verilog
integer flag, MyCount, I[0:63];
```

Real

A real declaration declares one or more variables of type real using IEEE STD-754-1985 (the IEEE standard for double precision floating point numbers). Arrays of reals can be declared using a range which defines the upper and lower indices of the array where the indices are constant expressions and shall evaluate to a positive or negative integer, or zero.

Example

```verilog
real X[1:10], Tox, Xj, Cgs;
```

Type Conversion

Verilog-A maintains the number type during expression evaluation and will also silently convert numbers to the type of the variable. This can lead to unexpected behavior. For example, the contribution statement,

```verilog
I(di,si) <+ white_noise(4 * 'P_K * T * (2/3) * abs(gm), "shot");
```

will always return 0 since the $2/3$ is evaluated using integer mathematics, and no noise is contributed from the noise power expression. Instead, use $2.0/3.0$ which will evaluate to a real number.
Data Types

Net Discipline

The net discipline is used to declare analog nets. A net is characterized by the discipline that it follows. Because a net is declared as a type of discipline, a discipline can be considered as a user-defined type for declaring a net.

A discipline is a set of one or more nature definitions forming the definition of an analog signal whereas a nature defines the characteristics of the quantities for the simulator. A discipline is characterized by the domain and the attributes defined in the natures for potential and flow.

The discipline can bind:

- One nature with potential
- Nothing with either potential or flow (an empty discipline)

System defined disciplines are predefined in the disciplines.vams file, a portion of which is shown below.

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

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

// Potential in volts
nature Voltage
units = "V";
```
Ground Declaration

A global reference node, or ground, can be associated with an already declared net of continuous discipline.

Syntax

```plaintext
ground list_of_nets;
```

Example

```plaintext`
`include "disciplines.vams"
module load(p);
    electrical p, gnd;
    ground gnd;
    parameter real R=50.0;
    analog
        V(p) <+ R * I(p, gnd);
endmodule
```

Implicit Nets

Nets used in a structural description do not have to be explicitly declared. The net is declared implicitly as scalar, the discipline as empty, and the domain as undefined.

Example

```plaintext`
`include "disciplines.vams"
module Implicit_ex(Input1, Input2, Output1, Output2, Output3);
    input  Input1, Input2;
    output Output1, Output2, Output3;
    electrical Input1, Input2, Output1, Output2, Output3;
    blk_a a1(Input1, a_b1);
    blk_a a2(Input2, a_b2);
    blk_b b1(a_b1, c_b1);
    blk_b b2 (a_b2, c_b2);
endmodule
```
Data Types

```
blk_c c1(Output1,Output2, Output3,c_b1,c_b2);
endmodule
```

**Genvar**

Genvars are used for accessing analog signals within behavioral looping constructs.

```plaintext
genvar list_of_genvar_identifiers;
```

where `list_of_genvar_identifiers` is a comma-separated list of genvar identifiers.

**Example**

```plaintext
genvar i, j;
```

**Parameters**

Parameters provide the method to bring information from the circuit to the model.

Parameter assignments are a comma-separated list of assignments. The right hand side of the assignment is a constant expression (including previously defined parameters).

For parameter arrays, the initializer is a list of constant expressions containing only constant numbers and previously defined parameters within `{ and } bracket delimiters.

Parameters represent constants; their values cannot be modified at runtime.

The general format is:

```plaintext
parameter {real | integer}list_of_assignments;
```

where the `list_of_assignments` is a comma-separated list of

```plaintext
parameter_identifier = constant {value-range}
```

where `value-range` is of the form

```plaintext
from value_rangeSpecifier
| exclude value_rangeSpecifier
| exclude constant_expression
```

where the `value_rangeSpecifier` is of the form

```plaintext
start_paren expression1 : expression2 end_paren
```
where start_paren is ‘{’ or ‘(‘ and end_paren is ‘}’ or ‘)’ and expression1 is constant_expression or ‘-inf’ and expression2 is constant_expression or ‘inf’.

The type (real | integer) is optional. If it is not given, it will be derived from the constant assignment value. A parenthesis indicates the range can go up to, but not include the value, whereas a bracket indicates the range includes the endpoint.

Example

    parameter real TestFlag = 0 from [0:inf) exclude (10:100) exclude (200:400);
    parameter real Temp = 27 from [-273.15:inf);
    parameter R = 50 from (0:inf);

and value ranges can have simple exclusions:

    parameter R = 50 from (0:inf] exclude (10:20) exclude 100;
Data Types
Chapter 5: Analog Block Statements

This chapter describes the procedural block. The analog block is where most of the analog behavior is described. This chapter will discuss the various procedural control statements available in Verilog-A as well as the use of events to control behavior.

Sequential Block

A sequential block is a grouping of two or more statements into one single statement. The format is:

```
begin [ : block_identifier { block_item_declaration } ]
{ statement }
end
```

The optional block identifier allows for naming of the block and local declaration of variables.

Example

```verilog
if (Vds < 0.0) begin: RevMode
    real T0; // T0 is visible in this block only
    T0 = Vsb;
    Vsb = Vsb + Vds;
    Vdss = - Vds + T0;
end
```

Conditional Statement (if-else)

The conditional statement is used to determine whether a statement is executed or not. The syntax is:

```
if ( expression ) true_statement;
[ else false_statement; ]
```

If the expression evaluates to True (non-zero), then the true_statement will be executed (or not, if false). If there is an else false_statement and the expression evaluates to False (zero), the false_statement is executed instead. Conditional statements may be nested to any level.
Analog Block Statements

Example

if (Vd < 0)
begin
  if (Vd < -Bv)
    Id = -Area * Is_temp * (limexp(-(Bv + Vd) / Vth) + Bv / Vth);
  else if (Vd == -Bv)
    Id = -Area * Ibv_calc;
  else if (Vd <= -5 * N * Vth)
    Id = -Area * Is_temp;
  else // -5 nKT/q <= Vd < 0
    Id = Area * Is_temp * (limexp(Vd / Vth) - 1);
  end
else
  Id = Area * Is_temp * (limexp(Vd / (N * Vth)) - 1);
end

Case Statement

A case statement is useful for multiple actions to be selected based on an expression. The format is:

```
case ( expression ) case_item { case_item } endcase
```

where case_item is:

- `expression { , expression } : statement_or_null`
- `default [ : ] statement_or_null`

The default-statement is optional; however, if it is used, it can only be used once. The case expression and the case_item expression can be computed at runtime (neither expression is required to be a constant expression). The case_item expressions are evaluated and compared in the exact order in which they are given. If one of the case_item expressions matches the case expression given in parentheses, then the statement associated with that case_item is executed. If all comparisons fail then the default item statement is executed (if given). Otherwise none of the case_item statements are executed.
Example

```literate
example(rgeo)
1, 2, 5: begin
  if (nuEnd == 0.0)
    Rend = 0.0;
  else
    Rend = Rsh * DMCG / (Weffcj * nuEnd);
end
3, 4, 6: begin
  if ((DMCG + DMCI) == 0.0)
    $strobe("(DMCG + DMCI) cannot be equal to zero\n");
  if (nuEnd == 0.0)
    Rend = 0.0;
  else
    Rend = Rsh * Weffcj / (3.0 * nuEnd * (DMCG + DMCI));
end
default:
  $strobe("Warning: Specified RGE = %d not matched (BSIM4RdsEndIso)\n", rgeo);
endcase
```

Repeat Statement

The `repeat()` statement execute a statement a fixed number of times. Evaluation of
the expression determines how many times the statement is executed. The syntax for
the `repeat()` statements is:

```literate
repeat ( expression ) statement
```

Example

```literate
repeat (devIndex - startIndex) begin
  devTemp = incrByOne(devTemp, offset);
end
```

While Statement

The `while()` looping executes a statement until an expression becomes False. If the
expression is False when the loop is entered, the statement is not executed at all. The
syntax for the `while()` statement is:

```literate
while ( expression ) statement
```
Analog Block Statements

Example

while(devTemp < T) begin
  devTemp = incrTemp(devTemp, offset);
end

For Statement

The for() statement controls execution of its associated statement(s) using an index variable. If the associated statement is an analog statement, then the control mechanism must consist of genvar assignments and genvar expressions (no use of procedural assignments and expressions are allowed).

for (procedural_assignment ; expression;
     procedural_assignment ) statement

If the for() loop contains an analog statement, the format is:

for (genvar_assignment; genvar_expression;
     genvar_assignment ) analog_statement

Note that the two are syntactically equivalent except that the executed statement is also an analog statement (with the associated restrictions).

Example

for (i = 0; i < maxIndex; i = i +1;) begin
  outReg[i] = getValue(i);
end
Chapter 6: Mathematical Functions and Operators

Verilog-A supports a range of functions and operators that may be used to form expressions that describe model behavior and to control analog procedural block flow. Return values from these functions are only a function of the current parameter value.

Unary/Binary/Ternary Operators

Arithmetic operators follow conventions close to the C programming language.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>*</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>&gt;</td>
<td>Relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>Relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Relational</td>
</tr>
<tr>
<td>!=</td>
<td>Logical equality</td>
</tr>
<tr>
<td>==</td>
<td>Logical equality</td>
</tr>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td>Bit-wise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bit-wise and</td>
</tr>
<tr>
<td>^</td>
<td>Bit-wise inclusive or</td>
</tr>
<tr>
<td>^=</td>
<td>Bit-wise exclusive or</td>
</tr>
<tr>
<td>^=~</td>
<td>Bit-wise equivalence</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>Left shift</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>Right shift</td>
</tr>
<tr>
<td>?:</td>
<td>Conditional</td>
</tr>
<tr>
<td>or</td>
<td>Event or</td>
</tr>
<tr>
<td>{[]}</td>
<td>Concatenation, replication</td>
</tr>
</tbody>
</table>
Mathematical Functions and Operators

**Arithmetic Operators**

The arithmetic operators are summarized in Table 6-2.

Table 6-2. Arithmetic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>a + b</td>
<td>a plus b</td>
</tr>
<tr>
<td>a - b</td>
<td>a minus b</td>
</tr>
<tr>
<td>a * b</td>
<td>a times b</td>
</tr>
<tr>
<td>a / b</td>
<td>a divided by b</td>
</tr>
<tr>
<td>a % b</td>
<td>a modulo b</td>
</tr>
</tbody>
</table>

See also “Precedence” on page 6-5 and “Arithmetic Conversion” on page 6-6.

**Relational Operators**

Table 6-3 defines and summarizes the relational operators.

Table 6-3. Relational operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>a is less than b</td>
</tr>
<tr>
<td>a &gt; b</td>
<td>a is greater than b</td>
</tr>
<tr>
<td>a &lt;= b</td>
<td>a is less than or equal to b</td>
</tr>
<tr>
<td>a &gt;= b</td>
<td>a is greater than or equal to b</td>
</tr>
</tbody>
</table>

The relational operators evaluate to a zero (0) if the relation is false or one (1) if the relation evaluates to true. Arithmetic operations are performed before relational operations.

**Examples**

```plaintext
a = 10;
b = 0;
a < b evaluates to false.
```
Logical Operators

Logical operators consist of equality operators and connective operators and are summarized in Table 6-4.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>a == b</td>
<td>a is equal to b</td>
</tr>
<tr>
<td>a != b</td>
<td>a is not equal to b</td>
</tr>
<tr>
<td>a &amp;&amp; b</td>
<td>a AND b</td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>!a</td>
<td>not a</td>
</tr>
</tbody>
</table>

Bit-wise Operators

Bit-wise operators perform operations on the individual bits of the operands following the logic described in the tables below.

Table 6-5. Bitwise 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>

Table 6-6. Bitwise 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>

Table 6-7. Bitwise 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>
Shift Operators

The shift operators shift their left operand either right (>>) or left (<<) by the number of bit positions indicated by their right operand, filling the vacated bit positions with zeros (0). The right operand is treated as an unsigned number.

Example

```plaintext
integer mask, new;
analog begin
    mask = 1;
    new = (mask << 4);
end
```

Conditional (Ternary) Operator

The conditional operator consists of three operands, separated by the operators ? (question mark) and : (colon).

```
expression1 ? expression2 : expression3
```

The `expression1` is first evaluated. If it evaluates to false (0) then `expression3` is evaluated and becomes the result. If `expression1` is true (any non-zero value), then `expression2` is evaluated and becomes the result.

Example

```plaintext
BSIM3vth0 = (BSIM3type == 'NMOS) ? 0.7 : -0.7;
```
Precedence

Table 6-10 shows the precedence order of the operators, with operators in the same row having equal precedence. Association is left to right with the exception of the conditional (ternary) operator, which associates right to left. Parentheses can be used to control the order of the evaluation.

<table>
<thead>
<tr>
<th>Operators</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - ! ~ (unary)</td>
<td>Highest</td>
</tr>
<tr>
<td>* / %</td>
<td></td>
</tr>
<tr>
<td>+ - (binary)</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>== !=</td>
<td></td>
</tr>
<tr>
<td>&amp; ~&amp;</td>
<td></td>
</tr>
<tr>
<td>^ ^~ ~^</td>
<td></td>
</tr>
<tr>
<td></td>
<td>~</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>? :</td>
<td>Lowest</td>
</tr>
</tbody>
</table>

Concatenation Operator

The concatenation operator is used for joining scalar elements into compound elements.

Example

```
parameter real taps[0:3] = {1.0, 2.0, 3.0, 4.0};
```

Expression Evaluation

The expression evaluation follows the order precedence described in Table 6-10. If the results of an expression can be determined without evaluating the entire expression, the remaining part of the expression is not evaluated, unless it contains analog expressions. This expression evaluation rule is known as short-circuiting.
Mathematical Functions and Operators

Arithmetic Conversion

Verilog-A performs automatic conversion of numeric types based on the operation. For functions that take integers, real numbers are converted to integers by rounding to the nearest integer, with ties rounded away from zero (0). For operators, a common data type is determined based on the operands. If either operand is real, the other operand is converted to real.

Examples

\[ a = 7.0 + 3; \]  // 3 becomes 3.0 and then the addition is performed, \( a = 10.0 \)
\[ a = 1 / 3; \]  // The result of this integer division is zero, \( a = 0. \)
\[ a = 7.0 + 1 / 3; \]  /* The 1/3 is evaluated by integer division, cast to 0.0
and added to 7.0, \( a = 7.0; \) */

Mathematical Functions

Verilog-A supports a wide range of functions to help in describing analog behavior. These include the standard mathematical functions, transcendental and hyperbolic functions, and a set of statistical functions.

Standard Mathematical Functions

The mathematical functions supported by Verilog-A are shown in Table 6-11.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Domain</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ln()</td>
<td>natural log</td>
<td>x&gt;0</td>
<td>real</td>
</tr>
<tr>
<td>log(x)</td>
<td>log base 10</td>
<td>x&gt;0</td>
<td>real</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential</td>
<td>x&lt;80</td>
<td>real</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root</td>
<td>x&gt;=0</td>
<td>real</td>
</tr>
<tr>
<td>min(x,y)</td>
<td>minimum of x and y</td>
<td>all x, y</td>
<td>if either is real, returns real, otherwise returns the type of x,y.</td>
</tr>
<tr>
<td>max(x,y)</td>
<td>maximum of x and y</td>
<td>all x, y</td>
<td>if either is real, returns real, otherwise returns the type of x,y.</td>
</tr>
<tr>
<td>abs(x)</td>
<td>absolute value</td>
<td>all x</td>
<td>same as x</td>
</tr>
</tbody>
</table>
For the \texttt{min()}, \texttt{max()}, and \texttt{abs()} functions, the derivative behavior is defined as:
\begin{itemize}
  \item \texttt{min}(x,y) is equivalent to (x < y) \ ? x : y
  \item \texttt{max}(x,y) is equivalent to (x > y) \ ? x : y
  \item \texttt{abs}(x) is equivalent to (x > 0) \ ? x : -x
\end{itemize}

**Transcendental Functions**

The transcendental functions supported by Verilog-A are shown in Table 6-12. All operands are integer or real and will be converted to real when necessary. The arguments to the trigonometric and hyperbolic functions are specified in radians. The return values are real.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Domain</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{sin}(x)</td>
<td>sine</td>
<td>all x</td>
<td>real</td>
</tr>
<tr>
<td>\texttt{cos}(x)</td>
<td>cosine</td>
<td>all x</td>
<td>real</td>
</tr>
<tr>
<td>\texttt{tan}(x)</td>
<td>tangent</td>
<td>x \neq n (pi/2), n is odd</td>
<td></td>
</tr>
<tr>
<td>\texttt{asin}(x)</td>
<td>arc-sine</td>
<td>-1 &lt;= x &lt;= 1</td>
<td></td>
</tr>
<tr>
<td>\texttt{acos}(x)</td>
<td>arc-cosine</td>
<td>-1 &lt;= x &lt;= 1</td>
<td></td>
</tr>
<tr>
<td>\texttt{atan}(x)</td>
<td>arc-tangent</td>
<td>all x</td>
<td></td>
</tr>
<tr>
<td>\texttt{atan2}(x,y)</td>
<td>arc-tangent of x/y</td>
<td>all x, all y</td>
<td></td>
</tr>
<tr>
<td>\texttt{hypot}(x,y)</td>
<td>sqrt(x^2 + y^2)</td>
<td>all x, all y</td>
<td></td>
</tr>
<tr>
<td>\texttt{sinh}(x)</td>
<td>hyperbolic sine</td>
<td>x &lt; 80</td>
<td></td>
</tr>
<tr>
<td>\texttt{cosh}(x)</td>
<td>hyperbolic cosine</td>
<td>x &lt; 80</td>
<td></td>
</tr>
<tr>
<td>\texttt{tanh}(x)</td>
<td>hyperbolic tangent</td>
<td>all x</td>
<td></td>
</tr>
<tr>
<td>\texttt{asinh}(x)</td>
<td>arc-hyperbolic sine</td>
<td>all x</td>
<td></td>
</tr>
</tbody>
</table>
Mathematical Functions and Operators

Statistical Functions

Verilog-A supports a variety of functions to provide statistical distributions. All parameters are real valued with the exception of seed_expression, an integer. The functions return a pseudo-random number, of type real, based on the distribution type. When a seed is passed to one of these functions, the seed is modified. The system functions return the same value for a given seed value.

The $random Function

The $random() function returns a new 32-bit random number each time it is called. The return type is a signed integer.

Note   The modulus operator, %, can be used to restrict the return value. For $b > 0, $random \% b will restrict the random number to \((-b+1) : (b-1)\).

Syntax

$random[( seed_expression )];

where

The optional seed_expression can be used to control the random number generation and must be a signed integer variable.

Example

integer seed_value, random_value;
random_value = $random;
// returns a value between -31 and 31.
random_value = $random(seed_value) \% 32;

Table 6-12. Transcendental Functions Supported by Verilog-A

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>acosh(x)</td>
<td>arc-hyperbolic cosine</td>
<td>x \geq 1</td>
</tr>
<tr>
<td>atanh(x)</td>
<td>arch-hyperbolic tangent</td>
<td>-1 \leq x \leq 1</td>
</tr>
</tbody>
</table>
The $rdist_uniform Function

The $rdist_uniform() function returns a uniform distribution across the range. The start and end real parameters bound the values returned. The start value must be smaller than the end value.

Syntax

$$\text{rdist_uniform}( \text{seed_expression}, \text{start_expression}, \text{end_expression} );$$

Example

```c
// Returns values between 0:10
random_value = $rdist_uniform(mySeed, 0, 10);
```

The $rdist_normal Function

The $rdist_normal() function returns a normal distribution of values around the mean_expression. The mean_expression parameter causes the average value of the return value to approach the mean_expression.

Syntax

$$\text{rdist_normal}( \text{seed_expression}, \text{mean_expression}, \text{stdev_expression} );$$

where

- stdev_expression determines the shape (standard deviation) of the density function.
- A mean_expression value of zero (0) and a stdev_expression of one (1) generates a Gaussian distribution. In general, larger numbers for stdev_expression spread out the returned values over a larger range.

Example

```c
// Returns a Gaussian distribution
random_value = $rdist_normal(mySeed, 0, 1);
```
The \$rdist_exponential Function

The \$rdist_exponential() function generates a distribution that follows an exponential.

Syntax

\$

\text{rdist}\_\text{exponential}( \text{seed\_expression}, \text{mean\_expression} );$

where

\text{mean\_expression} parameter causes the average value of the return value to approach the mean. The \text{mean\_expression} value must be greater than zero (0).

Example

\[
// \text{Exponential distribution approaching 1}
\]
\[
\text{random\_value} = \$\text{rdist\_exponential}(\text{mySeed, 1});
\]

The \$rdist_poisson Function

The \$rdist_poisson() function returns a Poisson distribution centered around the \text{mean\_expression} value (the \text{mean\_expression} parameter causes the average value of the return value to approach the \text{mean\_expression}).

Syntax

\$

\text{rdist}\_\text{poisson}( \text{seed\_expression}, \text{mean\_expression} );$

where

\text{mean\_expression} value must be greater than zero (0).

Example

\[
// \text{Distribution around 1}
\]
\[
\text{random\_value} = \$\text{rdist\_poisson}(\text{mySeed,1});
\]

The \$rdist_chi_square Function

The \$rdist_chi_square() function returns a Chi-Square distribution.

Syntax

\$

\text{rdist}\_\text{chi\_square}( \text{seed\_expression}, \text{degree\_of\_freedom\_expression} );$

where
The $rdist_t Function

The $rdist_t() function returns a Student's T distribution of values.

Syntax

$rdist_t( seed_expression, degree_of_freedom_expression );

where

degree_of_freedom_expression parameter helps determine the shape of the density function. Larger values spread the returned values over a wider range. The degree_of_freedom_expression value must be greater than zero (0).

Example

// Student's T distribution of 1.0
random_value = $rdist_t(mySeed,1.0);

The $rdist_erlang Function

The $rdist_erlang() function returns values that form an Erlang random distribution.

Syntax

$rdist_erlang( seed_expression, k_stage_expression, mean_expression );

where

mean_expression and k_stage_expression values must be greater than zero (0). The mean_expression parameter causes the average value of the return value to approach this value.

Example

// Erlang distribution centered around 5.0 with a range of 2.0.
random_value = $rdist_erlang(mySeed,2.0, 5.0);
Mathematical Functions and Operators

6-12 Statistical Functions
Chapter 7: Analog Operators and Filters

Analog operators have the same functional syntax as other operators and functions in Verilog-A, but they are special in that they maintain an internal state. This impacts how and where they may be used.

Because they maintain their internal state, analog operators are subject to several important restrictions. These are:

- Analog operators cannot be used inside conditional (if and case) or looping (for) statements unless the conditional expression is a genvar expression (genvars cannot change their value during the course of an analysis).
- Analog operators are not allowed in the repeat and while looping statements.
- Analog operators can only be used inside an analog block; they cannot be used inside user-defined analog functions.
- Under most cases, you cannot specify a null argument in the argument list of an analog operator.

Filters are analog functions that provide a means of modifying waveforms. A range of Laplace and Z-transform filter formulations are available. transition() and slew() are used to remove discontinuities from piecewise linear and piecewise continuous waveforms.

The limexp() operator provides a way to bound changes in exponential functions in order to improve convergence properties.

Tolerances

Most simulators use iterative approaches to solve the system of nonlinear equations, such as the Newton-Raphson algorithm. Some criteria is needed to indicate that the numerical solution is close enough to the true solution. Each equation has a tolerance defined and associated with it (in most cases a global tolerance is applied). However, the analog operators allow local tolerances to be applied to their equations.

Parameters

Some analog operators (Laplace and Z-transform filters) require arrays as arguments.

Examples
Analog Operators and Filters

```c
integer taps[0:3];
taps = {1, 2, 3, 4};
vout1 = zi_nd(vn, taps, {1});
vout2 = zi_nd(vn, {1, 2, 3, 4}, {1});
```

**Time Derivative Operator**

The time derivative operator, `ddt()`, computes the derivative of its argument with respect to time.

**Syntax**

```
ddt( expr )
```

where

`expr` is an expression with respect to which the derivative will be taken.

**Example**

```
I(n1,n2) <+ C * ddt{V(n1, n2)};
```

**Time Integrator Operator**

The time integrator operator, `idt()`, computes the time integral of its argument.

**Syntax**

```
idt( expr, [ic[,assert[,abstol]]] )
```

where

`expr` is an expression to be integrated over time.
`ic` is an optional expression specifying an initial condition.
`assert` is an optional integer expression that when true (non-zero), resets the integration.
`abstol` is a constant absolute tolerance to be applied to the input of the `idt()` operator and defines the largest signal level that can be considered to be negligible.

In DC analyses, the `idt()` operator returns the value of `ic` whenever `assert` is given and is true (non-zero). If `ic` is not given, `idt()` multiplies its argument by infinity for
DC analyses. So if the system does not have feedback that forces the argument to zero, ic must be specified.

Example

\[ V(\text{out}) \leftarrow \text{gain} \times \text{idt}(V(\text{in}) - V(\text{out}),0) + \text{gain} \times V(\text{in}); \]

**Circular Integrator Operator**

The circular integrator operator, \text{idtmod()}, converts an expression argument into its indefinitely integrated form.

**Syntax**

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

where

- \text{expr} is the expression to be integrated.
- \text{ic} is an optional expression specifying an initial condition. The default value is zero (0).
- \text{modulus} is a positive-valued expression which specifies the value at which the output of \text{idtmod()} is reset. If not specified, \text{idtmod()} behaves like the \text{idt()} operator and performs no limiting on the output of the integrator.
- \text{offset} is a dynamic value added to the integration. The default of \text{offset} is zero (0).

The modulus and offset parameters define the bounds of the integral. The output of the \text{idtmod()} function always remains in the range:

\[ \text{offset} \leq \text{idtmod\_output} < \text{offset} + \text{modulus} \]

Example

\[ \text{phase} = \text{idtmod}(\text{fc} + \text{gain} \times V(\text{in}), 0, 1, 0); \]

**Absolute Delay Operator**

The absolute delay operator, \text{absdelay()}, is used to provide delay for a continuous waveform.

**Syntax**

\[
\text{absdelay}( \text{expr}, \text{time\_delay}, \text{max\_delay} )
\]
Analog Operators and Filters

where

expr is the expression to be delayed

time_delay is a nonnegative expression that defines how much expr is to be delayed

If the optional max_delay is specified, the value of time_delay can change during a simulation, as long as it remains positive and less than max_delay. If max_delay is not specified, any changes to time_delay are ignored. If max_delay is specified and changed, any changes are ignored and the simulator will continue to use the initial value.

In DC and OP (operating point) analyses, absdelay() returns the value of expr. In AC and small-signal analyses, the input waveform expr is phase shifted according to:

\[ Y(\omega) = X(\omega)\cdot e^{-j\omega \cdot \text{time\_delay}} \]

In the time domain, absdelay() introduces a delay to the instantaneous value of expr according to the formula:

\[ y(t) = x(t - \text{time\_delay}) \quad \text{where time\_delay} \geq 0 \]

Example

\[ V\_\text{delayed} = \text{absdelay}( V(\text{in}), \text{time\_delay} ) \]

Transition Filter

The transition filter, transition(), is used to smooth out piecewise constant waveforms. The transition filter should be used for transitions and delays on digital signals as it provides controlled transitions between discrete signal levels. For smoothly varying waveforms, use the slew filter, slew().

Syntax

\[
\text{transition} \left( \text{expr}[,\text{time\_delay}[,,\text{rise\_time}[,,\text{fall\_time}[,,\text{time\_tol}]]]] \right)
\]

where all values are real and time_delay, rise_time, fall_time, and time_tol are optional and

expr is the input expression waveform to be delayed

time_delay is the delay time and must be \( \geq 0 \) (defaults to zero (0))

rise_time is the transition rise time and must be \( \geq 0 \)
fall_time is the transition the fall time and must be $\geq 0$ (If fall_time is not specified and rise_time is specified, the value of rise_time will be used)

time_tol is the absolute tolerance and must be $> 0$

The transition() filter forces all the positive transitions of the waveform expr to have a rise time of rise_time and all negative transitions to have a fall time of fall_time (after an initial delay of time_delay).

The transition() function returns a real number which describes a piecewise linear function. It forces the simulator to put time-points at both corners of a transition and to adequately resolve the transitions (if time_tol is not specified).

In DC analyses, the output waveform is identical to the input waveform expr. For AC analyses, the transfer function is modeled as having unity transmission across all frequencies.

Figure 7-1 shows an example of a transition() filter on a pulse waveform.

If interrupted on a rising transition, the function will attempt to finish the transition in the specified time with the following rules (see Figure 7-2):

- If the new time value is below the value at the time of the interruption, the function will use the old destination as the origin.
- If the new destination is above the value at the time of the interruption, the first origin is retained.
Slew Filter

The slew filter, `slew()`, provides a means to bound the rate of change of a waveform. A typical use of this analog operator would be to generate continuous signals from a piecewise continuous signal. Discrete-valued signals would use the `transition()` function.

Syntax

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

where all the arguments are real numbers and

- `expr` in the input waveform expression
- `max_pos_slew_rate` is the maximum positive slew rate allowed. `max_pos_slew_rate` is optional and must be > 0
- `max_neg_slew_rate` is the maximum negative slew rate allowed.

Any slope of the waveform `expr` that is larger than `max_pos_slew_rate` is limited to `max_pos_slew_rate` for positive transitions and limited to `max_neg_slew_rate` for negative transitions. If no rates are specified, `slew()` returns `expr` unchanged. If the slope of `expr` is in-between the maximum slew rates, the input `expr` is returned.

In DC analyses, the input `expr` is passed through the filter unchanged. In AC small-signal analyses, the `slew()` operator has a unity transfer function. In this case it has zero transmission.
Last Crossing Function

The last crossing function, \texttt{last\_crossing()}, is used to find where a signal expression last crossed zero (0).

Syntax

\begin{verbatim}
last_crossing( expr, dir )
\end{verbatim}

where

expr is the signal expression

dir is an integer flag with values -1, 0, +1

If \texttt{dir} is set to 0 or is not specified, the last crossing will be detected on both positive and negative signal crossings. If \texttt{dir} is +1 or -1, then the last crossing will only be detected on rising edge (falling edge) transitions of the signal.

If \texttt{expr} has not crossed zero for the first time, the function will return a negative value.

Limited Exponential

An alternative function to the \texttt{exp()} standard mathematical function is the \texttt{limexp()} function. The \texttt{limexp()} function is mathematically equivalent to the \texttt{exp()} function but the simulator keeps track of the value of the argument at the previous Newton-Raphson iteration and limits the amount of change from one iteration to another. The purpose of this function is to provide better convergence. The simulator
will not converge until the return value of \texttt{limexp()} equals the exponential for that input.

\textbf{Syntax}

\begin{verbatim}
limexp(arg);
\end{verbatim}

\textbf{Example}

\begin{verbatim}
Is = Is0 * limexp(Vj / \$vt);
\end{verbatim}

\section*{Laplace Transform Filters}

Laplace transform filters are used to implement lumped linear continuous-time filters.

\subsection*{laplace_zp()}

The \texttt{laplace_zp()} is used to implement the zero-pole form of the Laplace transform filter.

\textbf{Syntax}

\begin{verbatim}
laplace_zp( expr, \zeta, \rho )
\end{verbatim}

where

\texttt{expr} is the expression to be transformed.

\( \zeta \) (zeta) is a vector of \( M \) pairs of real numbers where each pair of numbers represents a zero. For each pair, the first number is the real part of the zero, the second number is the imaginary part.

\( \rho \) (rho) is a vector of \( N \) real pairs, one for each pole. The poles of the function are described in the same manner as the zeros (the first number is the real part, the second number is the imaginary part).

The transfer function is:

\begin{equation}
H(s) = \frac{\prod_{k=0}^{M-1} \left( 1 - \frac{s}{\zeta_k + j\zeta_k} \right)}{\prod_{k=0}^{N-1} \left( 1 - \frac{s}{\rho_k + j\rho_k} \right)}
\end{equation}
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. For a real pole or zero root, the imaginary term is specified as zero (0). If a root is complex, its conjugate must also be specified. If a root is zero (0), it is implemented as $s$, rather than $(1-s/r)$, where $r$ is the root.

**laplace_zd()**

The `laplace_zd()` represents the zero-denominator form of the Laplace transform filter.

**Syntax**

```plaintext
laplace_zd( expr, \zeta, d )
```

where

- `expr` is the expression to be transformed.
- $\zeta$ (zeta) is a vector of $M$ pairs of real numbers where each pair of numbers represents a zero. For each pair, the first number is the real part of the zero, the second number is the imaginary part.
- `d` is a vector of $N$ real numbers representing the coefficients of the denominator.

The transfer function is:

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

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. For a real zero, the imaginary term is specified as zero (0). If a root is complex, its conjugate must also be specified. If a root is zero (0), it is implemented as $s$, rather than $(1-s/r)$, where $r$ is the root.
Analog Operators and Filters

**laplace_np()**

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

**Syntax**

```python
laplace_np( expr, n, ρ )
```

where
- `expr` is the expression to be transformed.
- `n` is a vector of `M` pairs of real numbers containing the coefficients of the numerator.
- `ρ` (rho) is a vector of `N` pairs of real numbers. Each pair represents a pole, the first number in the pair is the real part of the pole and the second is the imaginary part.

The transfer function is:

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

where \( \rho_k^r \) and \( \rho_k^i \) are the real and imaginary parts of the \( k \)th pole and \( n_k \) is the coefficient of the \( k \)th power of \( s \) in the numerator. For a real pole, the imaginary term is specified as zero (0). If a pole is complex, its conjugate must also be specified. If a pole is zero (0), it is implemented as \( s \), rather than \((1-s/r)\), where \( r \) is the pole.

**laplace_nd()**

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

**Syntax**

```python
laplace_nd( expr, n, d )
```

where
expr is the expression to be transformed.

n is a vector of M pairs of real numbers containing the coefficients of the numerator.

d is a vector of N real numbers containing the coefficients of the denominator.

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.

Z-Transform Filters

The Z-transform filters implement linear discrete-time filters. Each filter uses a parameter $T$ which specifies the filter's sampling period. The zeros argument may be represented as a null argument. The null argument is produced by two adjacent commas (,,) in the argument list.

All Z-transform filters share three common arguments: $T$, $t$, and $t_0$.

- $T$ specifies the period of the filter, is mandatory, and must be positive.
- $t$ specifies the transition time, is optional, and must be non-negative.
- $t_0$ specifies the first transition time. If it is not supplied, the first transition is at $t=0$.

zi_zp()

The $\text{zi_zp}()$ operator implements the zero-pole form of the Z-transform filter.

Syntax

$$\text{zi_zp}( \text{expr}, \zeta, \rho, T[,t[,t_0]] )$$

where
Analog Operators and Filters

expr is the expression to be transformed.

ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero (0) and the second is the imaginary part.

ρ (rho) is a vector of N real numbers, one for each pole, represented in the same manner as the zeros.

The transfer function is:

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

where \( \zeta_k^r \) and \( \zeta_k^i \) are the real and imaginary parts of the kth zero, while \( \rho_k^r \) and \( \rho_k^i \) are the real and imaginary parts of the kth pole. If a root (a pole or zero) is real, the imaginary part must be specified as zero (0). If a root is complex, its conjugate must also be present. If a root is zero (0), the term associated with it is implemented as \( z \), rather than as \( (1 - z/r) \) where r is the root.

zi_zd()

The zi_zd() operator implements the zero-denominator form of the Z-transform filter.

Syntax

\[ zi\_zd(\text{expr}, \zeta, \text{d}, \text{T}[,\tau[,\text{t}_0]]) \]

where

expr is the expression to be transformed.

ζ (zeta) is a vector of M pairs of real numbers. Each pair of 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 vector of N real numbers containing the coefficients of the denominator.

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

**zi_np()**

The `zi_np()` implements the numerator-pole form of the Z-transform filter.

**Syntax**

```
zi_np( expr, n, \rho \{, \tau \{, t_0 \}\} )
```

where

- `expr` is the expression to be transformed.
- `n` is a vector of $M$ real numbers containing the coefficients of the numerator.
- `\rho` (rho) is a vector of $N$ pairs of real numbers where each pair represents a pole, the first number in the pair is the real part of the pole and the second is the imaginary part.

The transfer function is:

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

where $n_k$ is the coefficient of the $k^{th}$ power of $z$ in the numerator, while $\rho_k^r$ 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 zero (0). If a pole is complex, its conjugate must also be specified. If a pole is zero (0), then the term associated with it is implemented as $z$, rather than as $(1 - z/\rho)$ where $\rho$ is the pole.

$zi_{nd}()$

The $zi_{nd}()$ implements the numerator-denominator form of the Z-transform filter.

**Syntax**

$zi_{nd}( \text{expr, } n, d, T[, \tau, t_0] )$

where

- $\text{expr}$ is the expression to be transformed.
- $n$ is a vector of $M$ real numbers containing the coefficients of the numerator.
- $d$ is a vector of $N$ real numbers containing the coefficients of the denominator.

The transfer function is:

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

where $n_k$ is the coefficient of the $k^{th}$ power of $s$ in the numerator and $d_k$ is the coefficient of the $k^{th}$ power of $z$ in the denominator.
Chapter 8: Analog Events

The analog behavior of a component can be controlled using events. Events have the characteristics of no time duration and events can be triggered and detected in different parts of the behavioral model.

There are two types of analog events: global events and monitored events.

Global events. These events are the initial_step event and the final_step event.

Monitored events. These events are the cross() function and the timer() function.

Events are detected using the @ operator. Null arguments are not allowed.

Global Events

A global event can be generated by the simulator at various times during the simulation. A Verilog-A module cannot generate an event but can only detect them using an event expression. The two predefined global events are initial_step and final_step. These events are triggered at the initial (first) and final (last) point in an analysis.

The initial_step Event

The initial_step event is triggered at the first time point of an analysis.

Syntax

@initial_step [{list_of_analyses}]

where list_of_analyses is an optional comma separated list of quoted strings to be compared during the simulation.

An optional argument can specify a comma separated list of analyses for the active event. If a name matches the current analysis name, an event is triggered. If no list is given the initial_step global event is active during the first point (or during the initial DC analysis) of every analysis.

Example

@initial_step("tran","ac","dc")
The final_step Event

The final_step event is triggered at the last time point of an analysis.

Syntax

```java
@{final_step(list_of_analyses)}
```

where `list_of_analyses` is an optional comma separated list of quoted strings to be compared during the simulation.

An optional argument can specify a comma separated list of analyses for the active event. If a name matches the current analysis name, an event is triggered. If no list is given, the final_step global event is active during the last point of an analysis.

Example

```java
@{final_step("tran")}
```

Global Event Return Codes

Events provide a useful mechanism for executing code that should only occur at the first and last points of a simulation. Table 8-1 defines the return code for the particular event and analysis type.

<table>
<thead>
<tr>
<th>Analysis</th>
<th>DCOP OP</th>
<th>TRAN OP p1 pN</th>
<th>AC OP p1 pN</th>
<th>NOISE OP p1 pN</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial_step()</td>
<td>1</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>initial_step(&quot;ac&quot;)</td>
<td>0</td>
<td>0 0 0</td>
<td>1 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>initial_step(&quot;noise&quot;)</td>
<td>0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>1 0 0</td>
</tr>
<tr>
<td>initial_step(&quot;tran&quot;)</td>
<td>0</td>
<td>1 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>initial_step(&quot;dc&quot;)</td>
<td>1</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>initial_step(unknown)</td>
<td>0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
<tr>
<td>final_step()</td>
<td>0</td>
<td>0 1 0</td>
<td>0 1 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>final_step(&quot;ac&quot;)</td>
<td>0</td>
<td>0 0 0</td>
<td>0 0 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>final_step(&quot;noise&quot;)</td>
<td>0</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 0 1</td>
</tr>
<tr>
<td>final_step(&quot;tran&quot;)</td>
<td>0</td>
<td>0 1 0</td>
<td>0 0 0</td>
<td>0 0 0</td>
</tr>
</tbody>
</table>
In Table 8-1, \( p_1 \), \( p_N \) indicates the first and last points. OP indicates the Operating Point.

## Monitored Events

Monitored events are triggered due to changes in signals, simulation time, or other runtime conditions during the simulation.

## The \texttt{cross} Function

The \texttt{cross} function, \texttt{cross()}, is used for generating a monitored analog event. It is used to detect threshold crossings in analog signals when the expression crosses zero in the direction specified. The \texttt{cross()} function can control the timestep to accurately resolve the crossing. The format is:

\[
\text{cross( expr[,dir[,time_tol[,expr_tol]]])}
\]

where

- \( \text{expr} \) is a required argument
- \( \text{dir} \) is an optional argument that is an integer expression
- \( \text{time_tol} \) and \( \text{expr_tol} \) are optional arguments that are real

If the tolerances are not defined, they are set by the simulator. If either or both tolerances are defined, then the direction of the crossing must also be defined. The direction can only evaluate to +1, -1, or 0. If \( \text{dir} \) is set to 0 or is not specified, the event and timestep control will occur on both positive and negative signal crossings. If \( \text{dir} \) is +1, then the event and timestep control occurs on the rising transitions of the signal only. If \( \text{dir} \) is -1, then the event and timestep control occurs on the falling transitions of the signal only. For other transitions of the signal, the \texttt{cross()} function will not generate an event. The \( \text{expr_tol} \) and \( \text{time_tol} \) arguments represent the maximum allowable error between the estimated crossing point and the actual crossing point.
Analog Events

Example

The following description of a sample-and-hold illustrates how the `cross()` function can be used.

```verilog
module sample_and_hold (in, out, sample);
    output out;
    input in, sample;
    electrical in, out, sample;
    real state;
    analog begin
        @(cross(V(sample) -2.0, +1.0))
            state = V(in);
            V(out) <+ transition(state, 0, 10n);
    end
endmodule
```

The `cross()` function is an analog operator and shares the same restrictions as other analog operators. It cannot be used inside an `if()` or `case()` statement unless the conditional expression is a genvar expression. Also, `cross()` is not allowed in the `repeat()` and `while()` statements but is allowed in the analog `for()` statements.

The timer Function

The timer function, `timer()`, is used to generate analog events. It is used to detect specific points in time. The general form is:

```verilog
    timer (start_time[,period]);
```

where

- `start_time` is a required argument
- `period` and `time_tol` are optional arguments

The `timer()` function schedules an event to occur at an absolute time (`start_time`). If the `period` is specified as greater than zero, the timer function schedules subsequent events at all integer multiples of `period`.

Example

A pseudo-random bit stream generator is an example of how the `timer()` function can be used.

```verilog
    module bitStreamGen (out);
        output out;
        electrical out;
        parameter period = 1.0;
```
integer x;
analog begin
    @(timer(0, period))
    x = $random + 0.5;
    V(out) <+ transition( x, 0.0 );
end
endmodule

**Event or Operator**

The **or** operator provides a mechanism to trigger an event if any one of the events specified occurs.

**Example**

    @(initial_step or initial_step("static"))

**Event Triggered Statements**

When an event is triggered, the statement block following the event is executed. The statement block has two restrictions.

- The statements cannot have expressions that include analog operators.
- The statement can not be a contribution statement.
Analog Events
Chapter 9: Verilog-A and the Simulator

This chapter describes how to access information related to the simulator function as well as provide information to the simulator to control or support the simulator.

Environment Parameter Functions

The environment parameter functions return simulator environment information.

The temperature Function

The temperature function, $temperature(), returns the ambient temperature of the circuit in Kelvin. The function has no arguments.

Syntax

$temperature [{}]

Example

DevTemp = $temperature;

The abstime Function

The absolute time function, $abstime, returns the simulation time, in seconds.

Syntax

$abstime

Example

simTime = $abstime;

The realtime Function

The realtime function, $realtime, returns the simulation time in seconds.

Syntax

$realtime[{}]

Example

CurTimeIn_mS = $realtime();
Verilog-A and the Simulator

The Thermal Voltage Function

The thermal voltage function, $vt$, returns the thermal voltage ($kT/q$) at the circuit's ambient temperature. Optionally, a temperature (in Kelvin) can be supplied and the thermal voltage returned is calculated at this temperature.

**Syntax**

$vt[(temperature_expression)]$

**Example**

DevVth = $vt(Tnom + 'P_CELSIUS0); // Tnom in C

**Note** The macro $P_CELSIUS0$, defined in the constants.vams header file, provides a convenient way to offset temperatures.

Controlling Simulator Actions

Verilog-A supports several functions to allow the model code to influence the simulation flow.

Bounding the Time Step

The bound step function, $bound_step$, places a bound on the size of the next time step. The simulator may still choose to select a smaller time step but $bound_step$ will restrict the maximum step that will be used. The function has no return value.

**Syntax**

$bound_step(expression)$

where expression is a required argument and sets the maximum time step (in seconds) that the simulator will take.

**Example**

$bound_step(maxTimeStep);$
Announcing Discontinuities

The discontinuity function, `discontinuity`, provides information about discontinuities in the module. The function has no return value.

Discontinuities can cause convergence problems for simulators and should be avoided when possible. Filter functions such as `transition()`, `limexp()`, and others can be used to smooth behavior of discontinuous functions. It is not necessary to use the `discontinuity` function to declare discontinuities caused by switch branches and built-in system functions.

Syntax

`$discontinuity \[ (\text{constant_expression}) \];`

where `constant_expression` is an optional argument that indicates the degree of the discontinuity. That is, `$discontinuity(i)` implies a discontinuity in the `i`'th derivative of the constitutive equation taken with respect to the signal value or time; `i` must be non-negative.

Example

```latex
@\{\text{cross(V(input, output))}\}
$discontinuity(1); // Declare a discontinuity in slope
```

Analysis Dependent Functions

The analysis dependent functions interact with the simulator based on the analysis type.

Analysis Function

The analysis function, `analysis()`, provides a way to test the current analysis. The function accepts a single string or a list of strings as an argument and returns true (1) if any argument matches the current analysis type or false (0) if no matches are found.
Verilog-A and the Simulator

Syntax

```
analysis(analysis_list)
```

The analysis list is not predefined but is set by the simulator. Simulators typically support the analysis types defined by SPICE, see Table 9-1. If a type is unknown, the simulator returns no match. The return codes for analysis functions are summarized in Table 9-2.

### Table 9-1. Types of Analyses

<table>
<thead>
<tr>
<th>Name</th>
<th>Description of Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;ac&quot;</td>
<td>SPICE .AC analysis</td>
</tr>
<tr>
<td>&quot;dc&quot;</td>
<td>SPICE .OP (operation point) or DC</td>
</tr>
<tr>
<td>&quot;noise&quot;</td>
<td>SPICE .NOISE analysis</td>
</tr>
<tr>
<td>&quot;tran&quot;</td>
<td>SPICE .TRAN transient analysis</td>
</tr>
<tr>
<td>&quot;ic&quot;</td>
<td>SPICE .IC initial condition analysis which precedes a transient analysis</td>
</tr>
<tr>
<td>&quot;static&quot;</td>
<td>Equilibrium point analysis. Examples are DC analysis and other analyses that use a preceding DC analysis, such as AC or noise.</td>
</tr>
<tr>
<td>&quot;nodeset&quot;</td>
<td>Phase during static calculation where nodesets are forced</td>
</tr>
</tbody>
</table>

### Table 9-2. Analysis Function Return Codes

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Argument</th>
<th>DC</th>
<th>TRAN OP</th>
<th>TRAN AC</th>
<th>AC OP</th>
<th>AC AC</th>
<th>NOISE OP</th>
<th>NOISE AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>First part of &quot;static&quot;</td>
<td>&quot;nodeset&quot;</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>Initial DC state</td>
<td>&quot;static&quot;</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>Initial condition</td>
<td>&quot;ic&quot;</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>&quot;dc&quot;</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>Transient</td>
<td>&quot;tran&quot;</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>Small-signal</td>
<td>&quot;ac&quot;</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>&quot;noise&quot;</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>

Example

```
if (analysis("ic"))
    Vj = 0.7;
```
**AC Stimulus Function**

The AC stimulus function, `ac_stim()`, produces a sinusoidal stimulus for use during a small-signal analysis. During large-signal analyses such as DC and transient, the AC stimulus function returns zero (0). The small-signal analysis name depends on the simulator, but the default value is “ac”. If the small-signal analysis matches the analysis name, the source is activated with a magnitude of `mag` (default 1.0) and phase of `phase` (default 0.0, in radians).

**Syntax**

```plaintext
ac_stim([analysis_name [, mag [, phase ]]])
```

**Noise Functions**

A variety of functions provide a way to easily support noise modeling for small-signal analyses. Noise is not contributed for transient analyses. In these cases, use the `$random` system task to contribute noise.

**White Noise Function**

White noise processes are completely uncorrelated with any previous or future values, and are therefore frequency-independent.

**Syntax**

```plaintext
white_noise(pwr [, name])
```

generates a frequency-independent noise of power `pwr`. The optional `name` labels the noise contribution so that it can be grouped with other noise contributions of the same `name` in the same module when a noise contribution summary is produced.

**Example**

```plaintext
I(n1, n2) <+ V(n1, n2) / R + white_noise(4 * `P_K * $temperature / R, “thermal”);
```

**Flicker Noise Function**

The flicker noise function, `flicker_noise()`, models flicker noise processes.

**Syntax**

```plaintext
flicker_noise(pwr, exp [, name])
```
Verilog-A and the Simulator

generates a frequency-dependent noise of power \( pwr \) at 1 Hz which varies in proportion to the expression \( 1/f^{\text{exp}} \).

The optional name labels the noise contribution so that it can be grouped with other noise contributions of the same name in the same module when a noise contribution summary is produced.

Example

\[
I(n_1, n_2) \leftarrow \text{flicker\_noise}(K_F \ast \text{pow(abs}(I(n_1,n_2)), AF), 1.0, \text{"flicker"});
\]

Noise Table Function

The noise table function, \( \text{noise\_table()} \), provides a means to introduce noise via a piecewise linear function of frequency.

Syntax

\[
\text{noise\_table(vector [, name])}
\]

where

- vector contains pairs of real numbers such that the first number of each pair is frequency (in Hz) and the second is the noise power. The pairs must be specified in ascending frequencies. The \( \text{noise\_table()} \) function will linearly interpolate between number pairs in order to compute the power spectral density at each frequency.

- name is optional and labels the noise contribution so that it can be grouped with other noise contributions of the same name in the same module when a noise contribution summary is produced.

Example

\[
I(n_1, n_2) \leftarrow \text{noise\_table}({1,0.1, 100,0.2, 1e5,0.24}, \text{"surface"});
\]
Chapter 10: System Tasks and I/O Functions

This section lists the various system tasks and functions available to the user to access simulator analysis information and shows the usage. System functions provide access to system level tasks as well as access to simulator information.

File Input/Output Operations

There are two functions to provide reading and writing to files on the operating system, $fopen()$ and $fclose()$. These functions are not supported in the current release.

Display Output Operations

There are several functions available to display information to the user during a simulation. Each uses the same format specification but has slightly different modes of operation.

Strobe Function

The strobe function, $strobe()$, displays its argument when the simulator has converged for all nodes at that time point. The $strobe()$ function always appends a new line to its output. The $strobe()$ function returns a newline character if no arguments are passed.

Syntax

    $strobe(list_of_arguments);

Examples

    $strobe("The value of X is %g", X);
    $strobe(); // print newline

Display Function

The display function, $display()$, provides the same capability as the $strobe$ function but without the newline character.
System Tasks and I/O Functions

Syntax

\$
\text{display(list_of_arguments);}$

Example

\$
\text{display("
Warning: parameter X is \text{\%g}, max allowed is \text{\%g}\n\n", X, maxX);}$

Format Specification

The following tables describe the escape sequences available for the formatted output. The hierarchical format specifier, \%m, does not take an argument. It will cause the display task to output the hierarchical name of the module, task, function, or named block which invoked the system task using the hierarchical format specifier. This feature can be used to determine which module generated a message, in the case where many modules are instantiated.

<table>
<thead>
<tr>
<th>Table 10-1. Escape Sequences</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sequence</strong></td>
</tr>
<tr>
<td>\n</td>
</tr>
<tr>
<td>\t</td>
</tr>
<tr>
<td>\</td>
</tr>
<tr>
<td>*</td>
</tr>
<tr>
<td>\ddd</td>
</tr>
<tr>
<td>%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 10-2. Format Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Specifier</strong></td>
</tr>
<tr>
<td>%h or %H</td>
</tr>
<tr>
<td>%d or %D</td>
</tr>
<tr>
<td>%o or %O</td>
</tr>
<tr>
<td>%b or %B</td>
</tr>
<tr>
<td>%c or %C</td>
</tr>
<tr>
<td>%m or %M</td>
</tr>
<tr>
<td>%s or %S</td>
</tr>
</tbody>
</table>
Simulator Control Operations

Simulator control functions provide a means to interrupt simulator execution.

The $finish Simulator Control Operation

The finish task simulator control operation, $finish, forces the simulator to exit and optionally print a diagnostic message.

Syntax

```
$finish [ (n) ];
```

where n is an optional flag to either (0) print nothing, (1) print simulator time and location, or (2) print simulator time, location, and statistics. The default value is 1.

Example

```
if (myError)
    $finish(1);
```

The $stop Simulator Control Operation

The stop simulator control option, $stop, suspends the simulator at the converged timepoint and optionally prints a diagnostic message.

Syntax

```
$stop [ (n) ];
```

where n is an optional flag to either (0) print nothing, (1) print simulator time and location, or (2) print simulator time, location, and statistics. The default value is 1.

Table 10-3. Format Specifications for Real Numbers

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%e or %E</td>
<td>exponential format for real type</td>
</tr>
<tr>
<td>%f or %F</td>
<td>decimal format for real type</td>
</tr>
<tr>
<td>%g or %G</td>
<td>decimal or exponential format for real type using format that results in shorter printed output</td>
</tr>
</tbody>
</table>
System Tasks and I/O Functions

Example

```c
if (myError)
    $stop(1);
```
Chapter 11: The Verilog-A Preprocessor

Verilog-A provides a familiar set of language preprocessing directives for macro definitions, conditional compilation of code, and file inclusion. Directives are preceded by the accent grave (` ) character, which should not be confused with a single quote. The directives are:

- `define
- `else
- `ifdef
- `include
- `resetall
- `undef

Defining Macros

A macro is defined using the `define directive

```
`define name value
```

For example,

```
`define PI 3.14
```

defines a macro called PI which has the value 3.14. PI may now be used anywhere in the Verilog-A file after this definition. To use PI, the preprocessing directive character, accent grave (` ), must precede it. For example,

```
V(p,n) <+ sin(2*`PI*freq*time);
```

results in the following code

```
V(p,n) <+ sin(2*3.14*freq*time);
```

The directive name must be a valid identifier. It must be a sequence of alpha-numeric characters and underscores with a leading alpha character. Existing directive names cannot be used. This includes Verilog-A, Verilog-AMS and Verilog-2001 directives. Examples of invalid macro definitions are:

```
`define undef 1  // existing Verilog-A directive - wrong!
`define 1PICO ip // leading character invalid - wrong!
`define elsif 1  // Verilog 2001 directive - wrong!
```
Macro text may be presented on multiple lines by using the Verilog-A line continuation character, backslash (\), at the end of each line. The backslash must be the last character on the line. If white space is inserted after the continuation character then the system will not continue the line.

Macros may also be parameterized using an arbitrary number of arguments,

    `define name(arg1,arg2,arg3...)  value

For example,

    `define SUM(A,B)  A+B

defines a parameterized macro called SUM which may be subsequently used as

    V(out) <+ `SUM(V(in1),V(in2))

Argument names must also be valid identifiers and are separated by commas. There can be no space between the name of the macro and the first parenthesis. If there is a space, then the parenthesis and all characters that follow it are taken to be part of the macro definition text.

Macros may be re-defined. Doing so will produce a compiler warning. They may also be undefined using the `undef directive:

    `undef SUM

The `undef directive takes a single macro name as argument. Note that no directive character is used here. Using `undef on a macro that has not been defined results in a compiler warning.

All macros may be removed using the `resetall directive. This is not frequently used, as it effectively deletes all macros defined to this point in processing. The directive takes no arguments as

    `resetall

Including Files

The `include directive allows the inclusion of one file in another.

    `include “filename”

The `include directive accepts a single quoted string, a file name, as argument. If an absolute filename is given, the compiler looks for the referenced file. If a relative filename is given, the compiler first looks in the current working directory and then in the system include directory for the referenced file. In either case, if the file is
found, its contents are inserted into the current file in place of the include directive. If the file is not found then the system issues an error message. The system include directory is given by

```
$HPEESOF_DIR/tiburon-da/veriloga/include
```

Most Verilog-A files begin by including `disciplines.vams` and `constants.vams` as

```verilog
'include "disciplines.vams"
'include "constants.vams"
```

The compiler finds these system include files in the system include directory above. Include directives may be nested to twenty levels deep.

**Conditional Compilation**

Code may be conditionally compiled using the `ifdef-`else-`endif` preprocessor construct. For example,

```verilog
 ifdef macro
   statements
 else
   statements
 endif
```

If the conditional macro is defined, then the first set of statements are compiled, else the second set of statements are compiled. Both the true and false branches of the conditional must consist of lexicographically correct Verilog-A code. Note that as in `undef`, the preprocessing directive character is not used in the condition.

The `else` clause is optional and the construct may be written as,

```verilog
 ifdef macro
   statements
 endif
```

The following example performs output only if the `DEBUG` macro has been defined.

```verilog
ifdef DEBUG
   $strobe("Output Voltage:%e", V(out));
endif
```
Predefined Macros

The system has a number of predefined macros. The first is mandated by the Verilog-A standard. The macro `__VAMS_ENABLE__` is defined and has value 1.

Verilog-AMS and Verilog 1364 1995/2001 Directives

Verilog-AMS and Verilog 1364 directives are not available in the system, but they are all flagged as reserved directives for compatibility purposes. The directives are:

`default_discipline`
`celldefine`
`default_nettype`
`elsif`
`endcelldefine`
`ifndef`
`line`
`nounconnected_drive`
`timescale`
`unconnected_drive`

Defining a directive with one of the above names will result in a reserved directive error message.

Unsupported Directives

Verilog-A supports two additional directives, `default_transition` and `default_function_type_analog`. These directives are not supported in this release of the compiler.
Appendix A: Reserved Words in Verilog-A

This appendix lists the reserved Verilog-A keywords. It also includes Verilog-AMS and Verilog-2001 keywords which are reserved.

A
abs absdelay acos acosh ac_stim always analog analysis and asin asinh assign atan atan2 atanh

B,C
begin bound_step branch buf bufif0 bufif1 case casez ceil cmos connectrules cos cosh cross
d
ddt deassign default defparam delay disable discipline discontinuity driver_update
Verilog-A Keywords

### E
- `edge`  
- `else`  
- `end`  
- `endcase`  
- `endconnectrules`  
- `enddiscipline`  
- `endfunction`  
- `endmodule`  
- `endnature`  
- `endprimitive`  
- `endspecify`  
- `endtable`  
- `endtask`  

### F,G,H
- `floor`  
- `flow`  
- `for`  
- `force`  
- `forever`  
- `fork`  
- `from`  
- `function`  
- `generate`  
- `genvar`  
- `ground`  
- `highz0`  
- `highz1`  
- `hypot`  

### I,J
- `idt`  
- `idtmod`  
- `if`  
- `ifnone`  
- `inf`  
- `initial`  
- `initial_step`  
- `inout`  
- `input`  
- `integer`  
- `join`  

### L,M,N
- `laplace_nd`  
- `laplace_np`  
- `laplace_zd`  
- `laplace_zn`  
- `laplace_zo`  
- `min`  
- `module`  
- `nand`  
- `nor`  
- `or`  
- `xor`
<table>
<thead>
<tr>
<th>laplace_zp</th>
<th>nature</th>
</tr>
</thead>
<tbody>
<tr>
<td>large</td>
<td>negedge</td>
</tr>
<tr>
<td>last_crossing</td>
<td>net_resolution</td>
</tr>
<tr>
<td>limexp</td>
<td>nmos</td>
</tr>
<tr>
<td>ln</td>
<td>noise_table</td>
</tr>
<tr>
<td>log</td>
<td>nor</td>
</tr>
<tr>
<td>macromodule</td>
<td>not</td>
</tr>
<tr>
<td>max</td>
<td>notif0</td>
</tr>
<tr>
<td>medium</td>
<td>notif1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>O,P</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
</tr>
<tr>
<td>output</td>
</tr>
<tr>
<td>parameter</td>
</tr>
<tr>
<td>pmos</td>
</tr>
<tr>
<td>posedge</td>
</tr>
<tr>
<td>potential</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>R,S</th>
</tr>
</thead>
<tbody>
<tr>
<td>rcmos</td>
</tr>
<tr>
<td>real</td>
</tr>
<tr>
<td>realtime</td>
</tr>
<tr>
<td>reg</td>
</tr>
<tr>
<td>release</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>rmmos</td>
</tr>
<tr>
<td>rmos</td>
</tr>
<tr>
<td>rtran</td>
</tr>
<tr>
<td>rtranif0</td>
</tr>
<tr>
<td>rtranif1</td>
</tr>
<tr>
<td>scalared</td>
</tr>
</tbody>
</table>
### Verilog-A Keywords

#### T
- Table: `table`
- Tan: `tan`
- Tanh: `tanh`
- Task: `task`
- Temperature: `temperature`
- Time: `time`
- Timer: `timer`
- Tran: `tran`
- Tranif0: `tranif0`
- Tranif1: `tranif1`
- Transition: `transition`
- Tri: `tri`
- Tri0: `tri0`
- Tri1: `tri1`
- Triand: `triand`
- Trior: `trior`
- Trireg: `trireg`

#### V, W, X, Z
- Vectored: `vectored`
- Vt: `wt`
- Wait: `wait`
- Wand: `wand`
- Weak0: `weak0`
- Weak1: `weak1`
- While: `while`
- White noise: `white_noise`
- Wire: `wire`
- Wor: `wor`
- Wreal: `wreal`
- Xnor: `xnor`
- Xor: `xor`
- Zi nd: `zi nd`
- Zi np: `zi np`
- Zi zd: `zi zd`
- Zi zp: `zi zp`
Appendix B: Unsupported Elements

Table 11-1 lists the unsupported Verilog-A keywords and functionality.

Table 11-1. Unsupported Elements

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hierarchy:</strong></td>
<td>Ordered parameter lists in hierarchical instantiation</td>
</tr>
<tr>
<td></td>
<td>Hierarchical names, except for node.potential.abstol and node.flow.abstol, which are supported</td>
</tr>
<tr>
<td></td>
<td>Derived natures</td>
</tr>
<tr>
<td></td>
<td>The defparam statement</td>
</tr>
<tr>
<td><strong>Functions:</strong></td>
<td>Accessing variables defined in a function's parent module</td>
</tr>
<tr>
<td><strong>Input / Output:</strong></td>
<td>The %b format character</td>
</tr>
<tr>
<td></td>
<td>The 'ddd octal specification of a character</td>
</tr>
<tr>
<td></td>
<td>Parameter-sized ports</td>
</tr>
<tr>
<td></td>
<td>Enforcement of input, output, and inout</td>
</tr>
<tr>
<td><strong>System tasks:</strong></td>
<td>$realtime scaled to the `timescale directive</td>
</tr>
<tr>
<td></td>
<td>The %b, %o, and %h specifications</td>
</tr>
<tr>
<td></td>
<td>$monitor</td>
</tr>
<tr>
<td></td>
<td>File operations</td>
</tr>
</tbody>
</table>
Unsupported Elements
Appendix C: Standard Definitions

This appendix lists the current values of the standard header files that are part of the distribution.

The disciplines.vams File

```verbatim
/*
  Verilog-A definition of Natures and Disciplines
  $RCSfile: disciplines.vams,v $ $Revision: 1.1 $ $Date: 2003/09/22 01:36:17 $
*/
`ifdef DISCIPLINES_VAMS
  `else
  `define DISCIPLINES_VAMS 1
`endif
discipline logic
domain discrete;
enddiscipline

/*
* Default absolute tolerances may be overridden by setting the _ABSTOL prior to including this file
*/
// Electrical
// Current in amperes
nature Current
  units = "A";
  access = I;
  idt_nature = Charge;
  `ifdef CURRENT_ABSTOL
    abstol = `CURRENT_ABSTOL;
  `else
    abstol = 1e-12;
  `endif
endnature

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

`endif
endnature

// Potential in volts
nature Voltage
units = "V";
access = V;
idt_nature = Flux;
`ifdef VOLTAGE_ABSTOL
abstol = `VOLTAGE_ABSTOL;
`else
   abstol = 1e-6;
`endif
endnature

// Flux in Webers
nature Flux
units = "Wb";
access = Phi;
ddt_nature = Voltage;
`ifdef FLUX_ABSTOL
abstol = `FLUX_ABSTOL;
`else
   abstol = 1e-9;
`endif
endnature

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

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

// 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-12;
C-3
`endif
endnature

// Conservative discipline
discipline magnetic
potential Magneto_Motive_Force;
flow Flux;
enddiscipline

// Thermal

// Temperature in Kelvin
nature Temperature
units = "K";
access = Temp;
`ifdef TEMPERATURE_ABSTOL
abstol = `TEMPERATURE_ABSTOL;
`else
abstol = 1e-4;
`endif
endnature

// Power in Watts
nature Power
units = "W";
access = Pwr;
`ifdef POWER_ABSTOL
abstol = `POWER_ABSTOL;
`else
abstol = 1e-9;
`endif
endnature

// Conservative discipline
discipline thermal
potential Temperature;
flow Power;
enddiscipline

// Kinematic

// Position in meters
nature Position
units = "m";
access = Pos;
ddt_nature = Velocity;
`ifdef POSITION_ABSTOL
abstol = `POSITION_ABSTOL;
`else
abstol = 1e-6;
`endif
endnature
Standard Definitions

// 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

// Torque in Newtons
nature Angular_Force
Standard Definitions

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

The constants.vams File

/*
Verilog-A definition of Mathematical and physical constants
$RCSfile: constants.vams,v $ $Revision: 1.1 $ $Date: 2003/09/22 01:36:17 $
*/
ifdef CONSTANTS_VAMS
else
define CONSTANTS_VAMS 1
// M_ indicates a mathematical constant
define M_E 2.7182818284590452354
define M_LOG2E 1.4426950408889634074
define M_LOG10E 0.43429448190325182765
define M_LN2 0.69314718055994530942
define M_LN10 2.30258509299404568402
define M_PI 3.14159265358979323846
define M_TWO_PI 6.28318530717958647652
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_ indicates a physical constant
The compact.vams File

/*
Copyright 2002, 2003 Tiburon Design Automation, Inc. All rights reserved.
This software has been provided pursuant to a License Agreement
containing restrictions on its use. This software contains
valuable trade secrets and proprietary information of
Tiburon Design Automation, Inc. and is protected by law. It may
not be copied or distributed in any form or medium, disclosed
to third parties, reverse engineered or used in any manner not
provided for in said License Agreement except with the prior
written authorization from Tiburon Design Automation, Inc.
Useful, common macro definitions and utilities
$RCSfile: compact.vams,v $ $Revision: 1.1 $ $Date: 2003/09/22 01:36:17 $
*/

`ifdef COMPACT_VAMS
`else
`define COMPACT_VAMS 1
/*
SPICE-specific different values:
`define SPICE_GMIN 1.0e-12
`define SPICE_K 1.3806226e-23
`define SPICE_Q 1.6021918e-19
`define LARGE_REAL 1.0e38
`define MIN_CONDUCTANCE 1.0e-3
`define DEFAULT_TNOM 27
*/
Standard Definitions

/* NOT_GIVEN are codes that are used to detect if a parameter value has been passed (future extensions to Verilog-A should make this obsolete). */
`define NOT_GIVEN -9.9999e-99
`define INT_NOT_GIVEN -9999999
`define N_MINLOG 1.0e-38
`define MAX_EXP 5.834617425e14
`define MIN_EXP 1.713908431e-15
`define EXP_THRESHOLD 34.0
`define TRUE  1
`define FALSE 0

/* Useful macro for setting Type example: `SET_TYPE(P_TYPE, N_TYPE, Type); will set variable Type */
`define SET_TYPE(n, p, Type) Type = 1; if (p == 1) Type = -1; if (n == 1) Type = 1

/* Print out value:
example: `DEBUG_STROBE("myVariable", myVariable); */
`define DEBUG_STROBE(xName, x) 
  ifdef DEBUG
    $strobe("\n%s = %g", xName, 1.0*x) 
  else 
    $strobe(""") 
  endif
`endif

`endif
Appendix D: Condensed Reference

Verilog-A is an analog hardware description language standard from Open Verilog International (www.ovi.org). It can be used to describe analog circuit behavior at a wide range of abstraction from behavioral models of circuits to compact transistor model descriptions. The Verilog-A source code is compiled automatically, if necessary, during a simulation. The netlist format follows the conventional ADS netlisting scheme. Modules whose names match ADS components will automatically override the built-in model description.

Verilog-A Module Template

```verilog
`include "disciplines.vams" // Natures and disciplines
`include "constants.vams" // Common physical and math constants
module myModel(port1, port2);
electrical port1, port2;
parameter real input1= 1.0 from [0:inf];
parameter integer input2 = 1 from [-1:1] exclude 0;
real X;
// this is a comment
/* this is a
 * comment block */
analog begin
  @( initial_step ) begin
    // performed at the first timestep of an analysis
  end
  if (input2 > 0) begin
    $strobe("input2 is positive",input1)
    // module behavioral description
    V(port1, port2) <+ I(port1, port2) * input1;
  end
  @( final_step ) begin
    // performed at the last time step of an analysis
  end
endmodule
```
Data Types

Table 11-2. Data Types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Discrete numerical type</td>
</tr>
<tr>
<td></td>
<td>integer [integer_name {, integer_name};]</td>
</tr>
<tr>
<td>real</td>
<td>Continuous numerical type</td>
</tr>
<tr>
<td></td>
<td>real [real_name {, real_name...};]</td>
</tr>
<tr>
<td>parameter</td>
<td>Attribute that indicates data type is determined at module instantiation.</td>
</tr>
<tr>
<td></td>
<td>parameter parameter_type param_name = default_value [from [range_begin:range_end] [exclude exclude_value];]</td>
</tr>
</tbody>
</table>

Analog Operators and Filters

Analog operators and filters maintain memory states of past behavior. They can not be used in an analog function.

Table 11-3. Analog Operators and Filters

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time derivative</td>
<td>The ddt operator computes the time derivative of its argument.</td>
</tr>
<tr>
<td></td>
<td>ddt(expr)</td>
</tr>
<tr>
<td>Time integral</td>
<td>The idt operator computes the time-integral of its argument.</td>
</tr>
<tr>
<td></td>
<td>idt(expr, [ic [, assert [, abstol ]]])</td>
</tr>
<tr>
<td>Linear time delay</td>
<td>absdelay() implements the absolute transport delay for continuous waveform.</td>
</tr>
<tr>
<td></td>
<td>absdelay(input, time_delay [, maxdelay])]</td>
</tr>
</tbody>
</table>
### Table 11-3. Analog Operators and Filters

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Discrete waveform filters</strong></td>
<td>The transition filter smooths out piecewise linear waveforms.</td>
</tr>
<tr>
<td>slew( expr [ , max_pos_slew_rate [ , max_neg_slew_rate ] ] )</td>
<td>The last_crossing() function returns a real value representing the simulation time when a signal expression last crossed zero.</td>
</tr>
<tr>
<td>last_crossing(expr, direction)</td>
<td></td>
</tr>
<tr>
<td><strong>Laplace transform filters</strong></td>
<td>laplace_zd() implements the zero-denominator form of the Laplace transform filter. The laplace_np() implements the numerator-pole form of the Laplace transform filter. laplace_nd() implements the numerator-denominator form of the Laplace transform filter. laplace_zp() implements the zero-pole form of the Laplace transform filter.</td>
</tr>
<tr>
<td>laplace_zp(expr, z, r)</td>
<td></td>
</tr>
<tr>
<td><strong>Z-transform filters</strong></td>
<td>The Z-transform filters implement linear discrete-time filters. Each filter uses a parameter T which specifies the filter’s sampling period.</td>
</tr>
<tr>
<td>zi_zd()</td>
<td>The zeros argument may be represented as a null argument. The null argument is produced by two adjacent commas (,,) in the argument list.</td>
</tr>
<tr>
<td>zi_np()</td>
<td>All Z-transform filters share three common arguments: T, t, and t0. T specifies the period of the filter, is mandatory, and must be positive.</td>
</tr>
<tr>
<td>zi_nd()</td>
<td>t specifies the transition time, is optional, and must be nonnegative.</td>
</tr>
<tr>
<td>zi_zp( expr , z , r , T [ , t [ , t0 ] ] )</td>
<td>zi_zd() implements the zero-denominator form of the Z-transform filter. zi_np() implements the numerator-pole form of the Z-transform filter. zi_nd() implements the numerator-denominator form of the Z-transform filter. zi_zp() implements the zero-pole form of the Z-transform filter.</td>
</tr>
<tr>
<td><strong>limexp</strong></td>
<td>Limits exponential argument change from one iteration to the next.</td>
</tr>
<tr>
<td>limexp(expr)</td>
<td></td>
</tr>
</tbody>
</table>
Mathematical Functions

Table 11-4. Mathematical Functions Supported by Verilog-A

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Domain</th>
<th>Return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ln()</td>
<td>natural log</td>
<td>x&gt;0</td>
<td>real</td>
</tr>
<tr>
<td>log(x)</td>
<td>log base 10</td>
<td>x&gt;0</td>
<td>real</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential</td>
<td>X&lt;80</td>
<td>real</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root</td>
<td>x&gt;=0</td>
<td>real</td>
</tr>
<tr>
<td>min(x,y)</td>
<td>minimum of x and y</td>
<td>all x, y</td>
<td>if either is real, returns real, otherwise returns the type of x,y.</td>
</tr>
<tr>
<td>max(x,y)</td>
<td>maximum of x and y</td>
<td>all x, y</td>
<td>if either is real, returns real, otherwise returns the type of x,y.</td>
</tr>
<tr>
<td>abs(x)</td>
<td>absolute value</td>
<td>all x</td>
<td>same as x</td>
</tr>
<tr>
<td>pow(x,y)</td>
<td>(x^y)</td>
<td>if x&gt;=0, all y; if x&lt;0, int(y)</td>
<td>real</td>
</tr>
<tr>
<td>floor(x)</td>
<td>floor</td>
<td>all x</td>
<td>real</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>ceiling</td>
<td>all x</td>
<td>real</td>
</tr>
</tbody>
</table>
## Transcendental Functions

Table 11-5. Transcendental Functions Supported by Verilog-A

<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 != n (π/2), n is odd</td>
</tr>
<tr>
<td>asin(x)</td>
<td>arc-sine</td>
<td>-1 &lt;= x &lt;= 1</td>
</tr>
<tr>
<td>acos(x)</td>
<td>arc-cosine</td>
<td>-1 &lt;= x &lt;= 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>x &lt; 80</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>hyperbolic cosine</td>
<td>x &lt; 80</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 &gt;= 1</td>
</tr>
<tr>
<td>tanh(x)</td>
<td>arch-hyperbolic tangent</td>
<td>-1 &lt;= x &lt;= 1</td>
</tr>
</tbody>
</table>
**AC Analysis Stimuli**

Table 11-6. AC Analysis Stimuli

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC Stimulus</td>
<td>The AC stimulus function produces a sinusoidal stimulus for use during a small-signal analysis.</td>
</tr>
<tr>
<td></td>
<td>ac_stim([analysis_name [, mag [, phase]]])</td>
</tr>
</tbody>
</table>

**Noise Functions**

Table 11-7. Noise Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>White Noise</td>
<td>Generates a frequency-independent noise of power pwr.</td>
</tr>
<tr>
<td></td>
<td>white_noise(pwr [, name])</td>
</tr>
<tr>
<td>Flicker Noise</td>
<td>Generates a frequency-dependent noise of power pwr at 1 Hz which varies in proportion to the expression 1/f^exp.</td>
</tr>
<tr>
<td></td>
<td>flicker_noise(pwr, exp [, name])</td>
</tr>
<tr>
<td>Noise Table</td>
<td>Define noise via a piecewise linear function of frequency. Vector is frequency, pwr pairs in ascending frequencies.</td>
</tr>
<tr>
<td></td>
<td>noise_table(vector [, name])</td>
</tr>
</tbody>
</table>

**Analog Events**

Table 11-8. Analog Events

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial Step</td>
<td>Event trigger at initial step.</td>
</tr>
<tr>
<td></td>
<td>@(initial_step ([list_of_analyses]))</td>
</tr>
<tr>
<td>Final Step</td>
<td>Event trigger at final step.</td>
</tr>
<tr>
<td></td>
<td>@(final_step ([list_of_analyses]))</td>
</tr>
</tbody>
</table>
Cross Zero crossing threshold detection.
cross(expr [ , dir [ , time_tol [ , expr_tol]]]);

Timer Generate analog event at specific time.
timer ( start_time [ , period [ , time_tol ]]);

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross</td>
<td>Zero crossing threshold detection.</td>
</tr>
<tr>
<td></td>
<td>cross(expr [ , dir [ , time_tol [ , expr_tol]]]);</td>
</tr>
<tr>
<td>Timer</td>
<td>Generate analog event at specific time.</td>
</tr>
<tr>
<td></td>
<td>timer ( start_time [ , period [ , time_tol ]]);</td>
</tr>
</tbody>
</table>

**Timestep Control**

Table 11-9. Simulator Action Control Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundstep</td>
<td>Controls the maximum time step the simulator will take during a transient simulation.</td>
</tr>
<tr>
<td></td>
<td>$bound_step( expression );</td>
</tr>
<tr>
<td>Discontinuity</td>
<td>Provides the simulator information about known discontinuities to provide help for simulator convergence algorithms.</td>
</tr>
<tr>
<td></td>
<td>$discontinuity [{ constant_expression }];</td>
</tr>
</tbody>
</table>
Input/Output Functions

Table 11-10. Input/Output Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strobe</td>
<td>Display simulation data when the simulator has converged on a solution for all nodes using a printf() style format. $strobe(args)</td>
</tr>
<tr>
<td>Monitor</td>
<td>Same as $strobe but output only when a parameter changes. $monitor(args)</td>
</tr>
</tbody>
</table>

Simulator Environment Functions

The environment parameter functions return simulator environment information.

Table 11-11. Environment Parameter Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature</td>
<td>Return circuit ambient temperature in Kelvin. $temperature</td>
</tr>
<tr>
<td>Absolute Time</td>
<td>Return absolute time in seconds. $abstime</td>
</tr>
<tr>
<td>Real time</td>
<td>$realtime can have an optional argument which scales the time. If no argument is given, $realtime's return value is scaled to the <code>time_unit of the module which invoked it. If an argument is given, $realtime shall divide the absolute time by the value of the argument (i.e., scale to the value specified in the argument). The argument for $realtime follows the semantics of the </code>time_unit, that is it shall consist of an integer followed by a scale factor. Valid integers are: 1, 10, and 100; valid scale factors are: s (seconds), ms (milliseconds), us (microseconds), ns (nanoseconds), ps (picoseconds), and fs (femtoseconds). $realtime[ ( scale)];</td>
</tr>
</tbody>
</table>
### Module Hierarchy

Structural statements are used inside the module block but cannot be used inside the analog block.

```verilog
module_or_primative #( .param1(expr){ , .param2(expr}) ) instance_name
((node {, node});
```

---

### Table 11-11. Environment Parameter Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thermal voltage</td>
<td>$vt$ can optionally have $Temperature$ (in Kelvin) as an input argument and returns the thermal voltage ($kT/q$) at the given temperature. $vt$ without the optional input temperature argument returns the thermal voltage using $temperature$. $vt[{\text{Temperature}}]$</td>
</tr>
<tr>
<td>Analysis</td>
<td>Returns true (1) if current analysis matches any one of the passed arguments. $analysis(str {, str})$</td>
</tr>
</tbody>
</table>
Index

Symbols
$ (dollar sign), 3-4, 3-5
$abstime function, 9-1
$bound_step function, 9-2
$discontinuity function, 9-3
$display function, 10-1
$finish function, 10-3
$random function, 6-8, 9-5
$rdist_chi_square function, 6-10
$rdist_erlang function, 6-11
$rdist_exponential function, 6-10
$rdist_normal function, 6-9
$rdist_poisson function, 6-10
$rdist_t function, 6-11
$rdist_uniform function, 6-9
$realtime function, 9-1
$stop function, 10-3
$strobe function, 10-1
$temperature function, 9-1
$vt function, 9-2
% (modulo), 6-8
%m (hierarchical format specifier), 10-2
) (right parenthesis), 4-5
@ (at-sign operator), 8-1
_ (underscore), 3-3, 3-4
__VAMS_ENABLE__ (predefined macro), 11-4
` (accent grave), 3-5, 11-1
`celldefine (reserved directive), 11-4
`default_discipline (reserved directive), 11-4
`default_function_type_analog (unsupported directive), 11-4
`default_nettype (reserved directive), 11-4
`default_transition (unsupported directive), 11-4
`define (compiler directive), 11-1
`else (compiler directive), 11-1, 11-3
`elsif (reserved directive), 11-4
`endcelldefine (reserved directive), 11-4
`endif (compiler directive), 11-3
`ifdef (reserved directive), 11-4
`ifndef (reserved directive), 11-4
`include (compiler directive), 11-1, 11-2
`line (reserved directive), 11-4
`nounconnected_drive (reserved directive), 11-4
`resetall (compiler directive), 11-1, 11-2
`timescale (reserved directive), 11-4
`unconnected_drive (reserved directive), 11-4
`undef (compiler directive), 11-1, 11-2
{ (left bracket), 4-4
} (right bracket), 4-4, 4-5

A
absolute delay operator, 7-3
absolute tolerance, 7-2, 7-5
AC analyses, 7-4, 7-5
ac_stim function, 9-5
access functions, 2-3, 2-5
access function value, 2-2
AHDL, 1-4
ambient temperature, 9-1, 9-2
analog behavior, 2-2, 5-1, 6-6, 8-1
analog block, 2-2, 2-4, 5-1, 7-1
defining the analog block, 2-1, 2-2
analog events, 8-3, 8-4
analog expressions, 6-5
analog functions, 2-4, 2-5, 2-8
analog modeling, 1-1
analog operators, 7-1, 7-6, 8-4, 8-5
analog procedural block flow, 6-1
analog signals, 1-2, 2-3, 4-2, 8-3
analog statements, 5-4
analysis dependent functions, 9-3
arithmetic conversion, 6-6
arithmetic operators, 6-2
arrays, 4-1, 4-4, 7-1
assignment order, 2-6
attributes, 4-2
automatic conversion of numeric types, 6-6

B
backslash, 11-2
backward compatibility, 3-5
behavioral description, 2-1
behavioral model, 8-1
binary operators, 3-2
bit-wise operators, 6-3
block item declaration, 2-5
block naming, 5-1
block statements, 3-2
bounds, 9-2
branch contribution operator, 2-4
branch contribution statement, 2-4
branches, 2-3

C
case sensitivity, 3-4
case statements, 5-2, 7-1
Chi-Square distribution, 6-10
circular integrator operator, 7-3
comma-separated lists, 4-4
comments, 3-2
compact models, 1-2, 1-6
compiler directives, 3-5
'define, 11-1
'else, 11-1, 11-3
'endif, 11-1, 11-3
'include, 11-1, 11-2
'resetall, 11-1, 11-2
'undef, 11-1, 11-2
compiler technology, 1-4
complex constants, 3-3
compound elements, 6-5
concatenation operator, 6-5
conditional compilation, 11-1, 11-3
conditional execution, 2-5
conditional expressions, 7-1
conditional macros, 11-3
conditional operators, 3-2, 6-4
conditional statements, 5-1, 7-1
constant assignment value, 4-5
constant expressions, 4-1, 4-4, 5-2
constant numbers, 3-3, 4-4
constants, 3-3, 4-4
continuous discipline, 4-3
continuous domain, 2-3
contribution operator, 2-3
contribution statements, 2-3, 2-5, 8-5
delays on digital signals, 7-4
discontinuities, 9-3
discrete signals, 7-4, 7-6
domain, 4-2, 4-3
double quotes, 3-2

E
empty discipline, 4-2
endpoint, 4-5
environment parameter functions, 9-1
Erlang random distribution, 6-11
error messages, 11-3, 11-4
escape sequences, 10-2
event or operator, 8-5
events, 5-1, 8-1, 8-2
event expressions, 8-1
event triggering, 8-5
exponential, 6-10
functions, 7-1
expressions, 6-1, 8-5
expression evaluation, 4-1, 6-5

F
files
constants.vams, 9-2, 11-3
disciplines.vams, 4-2, 11-3, C-1
file inclusion, 11-1
filters, 7-1
Laplace transform, 7-8
laplace_nd, 7-10
laplace_np, 7-10
indices, 4-1
indirect branch assignment, 2-4
initial condition, 7-2, 7-3
initial_step global event, 8-1
input parameters, 2-5
instance
declaration, 2-6, 2-7
parameters, 2-7
instantiating
a module, 2-6
integers, 3-3, 4-1, 6-6, 6-7, 8-3
integration, 7-2
intellectual property protection, 1-4
interative approaches, 7-1

J
Jacobian, 1-5

K
k-stage, 6-11
keywords, 3-1, 3-4, 3-5
Kirchoff
voltage and current laws, 1-4

L
language
constructs, 3-4
preprocessing directives, 11-1
languages
C, 1-2, 6-1
HDL, 1-1
mixed-signal, 1-2
Verilog, 1-2
Verilog-A, 1-2
VHDL, 1-2
Laplace filters, 7-1, 7-8
large-signal analysis, 9-5
last_crossing function, 7-7
lexical conventions, 3-1, 11-3
limexp function, 9-3
line continuation character, 3-1
linear discrete-time filters, 7-11
local
declaration of variables, 5-1
tolerances, 7-1
variables, 2-5
logical operators, 6-3
looping statements, 7-1

M
macros, 11-1, 11-2
macro definitions and utilities, 11-1, C-7
predefined, 11-4
magnitude, 9-5
mathematical
equations, 1-2, 1-6
functions, 6-6
maximum slew rate, 7-6
mixed-signal language, 1-2
models, 1-2, 1-6, 6-1
modules, 2-8
module behavior, 9-3
module boundary, 2-8
module declaration, 2-1, 2-2, 2-6
module instantiation
resistor example, 2-1
module port definition, 2-7
modulus, 7-3
monitored events, 8-1, 8-3
multiple actions, 5-2

N
name spaces, 2-8
named blocks, 2-5, 2-8
natures, 4-2, C-1
negative transitions, 7-5
net_discipline, 4-2
nets, 2-3, 2-4, 4-3
analog nets, 4-2
implicit nets, 4-3
neural nets, 1-2
newline character, 10-1
newlines, 3-1, 3-2
Newton-Raphson, 1-5, 7-1
nodes, 1-4, 2-3, 2-4, 10-1
noise, 4-1
noise contribution, 9-5, 9-6
noise modeling, 9-5
noise power, 9-6
noise_table function, 9-6
nonlinear equations, 7-1
normal distribution, 6-9
null arguments, 7-1, 8-1
numbers, 3-1, 3-3
floating point, 3-3, 4-1
integer, 3-3
number type, 4-1

Index-4
real, 3-3
scale factors, 3-3

O
OP analyses, 7-4
operands, 6-6, 6-7
operating system, 10-1
operators, 6-1
absdelay, 7-3
arithmetic, 6-2
binary (double), 3-2
bit-wise, 6-3
concatenation, 6-5
conditional, 6-4
conditional (ternary), 3-2
ddt, 7-2
event or, 8-5
idt, 7-2
idtmod, 7-3
limexp, 7-1
logical, 6-3
precedence, 6-5
relational, 6-2
shift, 6-4
unary (single), 3-2
optimization, 1-1
optional block identifier, 5-1
ordered lists, 2-6, 2-7

P
parameters, 4-4
default parameter values, 2-6
defining parameters, 2-1, 2-2, 2-4
input parameters, 2-5
instance parameters, 2-7
parameter declaration, 2-6
parameter description, 1-1
parameter extraction, 1-3
parameterized macros, 11-2
seed_expression, 6-8
partial derivatives, 1-5
passed variable arguments, 2-5
phase, 9-5
phase shift, 7-4
piecewise
constant waveforms, 7-4
continuous signal, 7-6
linear, 7-5, 9-6
Poisson distribution, 6-10
ports
defining ports, 2-1, 2-2
port assignment, 2-7
positive transitions, 7-5
potential, 2-3, 4-2
power, 9-5
power spectral density, 9-6
predefined macros, 11-4
predefined mathematical constants
M_1_PI, C-6
M_2_PI, C-6
M_2_SQRTPI, C-6
M_E, C-6
M_LN10, C-6
M_LN2, C-6
M_LOG10E, C-6
M_LOG2E, C-6
M_PI, C-6
M_PI_2, C-6
M_PI_4, C-6
M_SQRT1_2, C-6
M_SQRT2, C-6
M_TWO_PI, C-6
predefined non-escaped identifiers, 3-4
predefined physical constants
P_C, C-7
P_CELSIUS0, 9-2, C-7
P_EPS0, C-7
P_H, C-7
P_K, C-7
P_Q, C-7
P_U0, C-7
preprocessing directives, 11-1
procedural block, 5-1
procedural control statements, 5-1
procedural statements, 2-2
pseudo-random bit stream generator, 8-4
pseudo-random numbers, 6-8

R
radians, 6-7, 9-5
random number generation, 6-8
range, 4-5
real
costants, 3-3
data types, 4-1
declarations, 4-1

Index-5
numbers, 3-3
reference net, 2-4
relational operators, 6-2
relative filenames, 11-2
repeat statements, 5-3, 7-1
reserved directives, 11-4
`celldesign, 11-4
`default discipline, 11-4
`default nettype, 11-4
`elsif, 11-4
`endcelldefine, 11-4
`ifndef, 11-4
`line, 11-4
`nounconnected_drive, 11-4
`timescale, 11-4
`unconnected_drive, 11-4
scalar, 4-3
elements, 6-5
scale factors, 3-3
scientific notation, 3-3
scope, 2-8
seed_expression, 6-8
sequential block, 6-5
signal
  access functions, 2-4
data, 2-3
signal-flow
  branch, 2-3
net, 2-3
signed integers, 6-8
simulation
  simulation flow, 9-2
  simulator functions, 9-1
single line comment, 3-2
slew filter, 7-4, 7-6
time, 1-4
$abstime, 9-1
$realtime, 9-1
time delay, 7-4
time derivative operator, 7-2
time duration, 8-1
time integral, 7-2
time integrator operator, 7-2
timer function, 8-4
timesteps, 8-3, 9-2
tokens, 3-1, 3-2
transcendental functions, 6-6, 6-7
transient analysis, 9-5
transition filter, 7-4, 7-6, 9-3
trigonometric functions, 6-7
type, 4-5
type conversion, 4-1
unary operators, 3-2, 3-3
underscore character, 3-3
uniform distribution, 6-9
unsupported directives
  `default_function_type_analog, 11-4
  `default_transition, 11-4
user-defined functions, 2-4, 7-1
tasks, 3-5
types, 4-2
variables, 1-6, 4-1
varying waveforms, 7-4
vectors, 2-3, 9-6

Student's T distribution of values, 6-11
system include
  directory, 11-3
  files, 11-3
system tasks and functions, 3-5
  system equations, 7-1
  system functions, 6-8

tabs, 3-1, 3-2
tasks, 2-8
temperature, 9-1
thermal voltage, 9-2
threshold crossings, 8-3
time, 1-4
$abstime, 9-1
$realtime, 9-1
time delay, 7-4
time derivative operator, 7-2
time duration, 8-1
time integral, 7-2
time integrator operator, 7-2
timer function, 8-4
timesteps, 8-3, 9-2
tokens, 3-1, 3-2
transcendental functions, 6-6, 6-7
transient analysis, 9-5
transition filter, 7-4, 7-6, 9-3
trigonometric functions, 6-7
type, 4-5
type conversion, 4-1
unary operators, 3-2, 3-3
underscore character, 3-3
uniform distribution, 6-9
unsupported directives
  `default_function_type_analog, 11-4
  `default_transition, 11-4
user-defined functions, 2-4, 7-1
tasks, 3-5
types, 4-2
variables, 1-6, 4-1
varying waveforms, 7-4
vectors, 2-3, 9-6

Index-6
Verilog, 1-2
Verilog 1364, 11-4
Verilog-2001, 11-1
Verilog-A, 1-2
  line continuation character, 11-2
modules, 2-1
Verilog-AMS, 11-1, 11-4
  keywords, 3-4
VHDL, 1-2
voltage contribution statement, 2-2

W
warnings, 11-2
waveforms, 7-5, 7-6
while statements, 5-3, 7-1
white noise, 9-5
white space, 3-1, 3-2, 11-2

Z
zero transmission, 7-6
zero-crossing, 1-6
Z-transform filters, 7-1, 7-11