Verilog-A Reference Manual
Table of Contents

Chapter 1: Introduction ................................................................. 6
    Analog Modeling ................................................................. 6
    Hardware Description Languages ........................................... 6
    Verilog-A ................................................................................ 7
    Systems ................................................................................ 7
        Conservative Systems ....................................................... 8
        Natures and Disciplines ..................................................... 8
        Signal Flow System .......................................................... 9
    Conventions Used in this Document ....................................... 9

Chapter 2: Verilog-A Modules ...................................................... 11
    Declaring Modules ............................................................. 11
        Module Definition ......................................................... 11
    Ports ..................................................................................... 12
    Describing Analog Behavior ............................................... 13
    Branches .............................................................................. 13
    Analog Signals ................................................................. 14
        Accessing Net and Branch Signals .................................. 14
        Indirect branch assignment .......................................... 14
    Branch Contribution Statement .......................................... 15
    Switch Branches .................................................................. 15

Hierarchical Structures .............................................................. 17
    Module Instance Parameter Value Assignment .................. 17
        By Order ........................................................................ 17
        By Name ........................................................................ 18
    Paramsets ........................................................................... 18
    Ports Assignments ............................................................ 21
    Port Assignment ............................................................... 21
        By Order ........................................................................ 21
        Hierarchical system parameters .................................... 22
    Scope ................................................................................... 23

User-defined Analog Functions .................................................. 23

Chapter 3: Lexical Conventions ................................................... 26
    White Space ......................................................................... 26
    Comments ........................................................................... 26
    Operators ............................................................................ 2
    Strings ................................................................................ 2
    Numbers .............................................................................. 2
        Integer Numbers ........................................................... 2
        Real Numbers .............................................................. 3
        Scale Factors .................................................................. 3
    Keywords ............................................................................. 4
    Identifiers ........................................................................... 4
    Escaped Identifiers ............................................................ 4
    System Tasks and Functions ................................................. 5
    Compiler Directives ............................................................. 5
    Attributes ............................................................................ 5

Chapter 4: Data Types ................................................................. 7
    Integer ................................................................................. 7
    Real .................................................................................... 7
Chapter 1: Introduction

This chapter introduces the Verilog-A language and software in terms of its capabilities, benefits, and typical use. Definitions of terms and conventions used in the document are described.

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.
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.

**Systems**

Verilog-A can be used to describe a system, a collection of interconnected components which are acted upon by a stimulus and produce a response, see figure 1. The components can be systems as well, this is called a hierarchical system. A component that does not include sub-components is a primitive component. Primitive components connect to zero or more nets, where each net connects to a signal. The component's behavior is defined in terms of the values at each of these nets.

A signal is contiguous collection of nets which may be hierarchical. Signals may be in the discrete domain, or digital signals, or in the continuous domain, or analog signals. A signal consisting of nets from both domains is called a mixed-signal. Verilog-A supports signals in the continuous domain, or analog signals.
Conservative Systems

Conservative systems have two values associated with every node in the system: the potential and the flow. In electrical systems these are the potential, or voltage across, and the flow through, or the current. Nodes follow Kirchoff’s Potential and Flow Laws. A branch is the path of flow between two nodes. Every branch has an associated potential across it and a flow through it.

In Verilog-A, if only one node is specified, the associated reference node is ground. The reference direction is defined such that flow is from positive to negative.

Natures and Disciplines

Verilog-A uses a set of disciplines to declare the types of nodes, nets, and branches. Disciplines specifies the natures for the potential and/or the flow of a branch, where a nature is a collection of attributes such as tolerances and access functions. The natures and disciplines are defined in a header file, disciplines.vams, that is typically included in each Verilog-A source file.
Signal Flow System

A signal flow system has only a potential nature binding specified. Signal flow models may be written such that the output of the modules are functions only of the potentials at the inputs of the module without taking flow into account. As such, it does not follow Kirchoff’s Flow Law (and so charge is not conserved).

It is possible that a single node may be bound to a number of nets of different disciplines. If a node is bound only to disciplines with a potential nature, then current contributions to that node are not allowed.

Conventions Used in this Document

This document is organized by chapter, each of which concentrates on one aspect of the language. The document is designed to illustrate the implementation of the language in practical uses and should be accompanied by the Verilog-AMS Language Reference Manual for completeness, especially for formal syntax.

In this document, the syntax is described in a compact readable form.

Lower case words are used to denote syntactic categories:

    block_statement

Bold faced words indicate reserved keywords, operators, and punctuation marks that are required parts of the syntax. Non-bold punctuation is used to denote choices or optional and repeating terms.

For example:

    endmodule;

Italicized words indicate terms and categories that will be defined further in the text. For example

    module my_module ;

indicates that the name of the module is my_module.

A vertical bar indicates alternative items. For example

    real | integer
Square brackets indicate optional items. For example

```
parameter [ real | integer ]
```

indicates that the keyword `parameter` is required but the keyword choice of `real` or `integer` is optional.

Braces indicate a repeated item. For example:

```
ground gnd { , pwr_gnd };
```

indicate that the list of items can optionally repeat, with a comma separating terms.
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.

Module Definition

A module definition is enclosed by the keywords module (or interchangeably, macromodule) and endmodule. The definition line also includes the port names and order, if any. The module definition can contain at most one analog block within the module statements. Module instantiation (which allows one module to incorporate another module into itself) is described later in this chapter.

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, and module_statements describe the module behavior and declarations of port types, analog functions, parameters, and variables.

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.

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.

### Syntax

```verilog
module module_name [ (port_expression { , port_expression } ) ] ;
```

where the `port_expression` is:

- a simple net identifier
- a scalar member of a vector net or port declared within the module
- a sub-range of a vector net or port declared within the module.

### Examples

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

...  
module modName(inPort, outPort);
    input inPort;
```
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

\[
\text{analog } \text{block\_statement}\]

where \text{block\_statement} is a single analog statement of a group of statements.

Examples

\[
\text{analog } V(n1, n2) <+ 1; // A simple 1 volt source}
\]

\[
\text{analog begin // A multi-statement analog block}
\]

\[
\text{vin} = V(\text{in});
\]

\[
\text{if } (\text{vin} >= \text{signal\_in\_dead\_high})
\]

\[
\text{vout} = \text{vin} - \text{signal\_in\_dead\_high};
\]

\[
\text{else}
\]

\[
\text{if } (\text{vin} <= \text{signal\_in\_dead\_low})
\]

\[
\text{vout} = \text{vin} - \text{signal\_in\_dead\_low};
\]

\[
\text{else}
\]

\[
\text{vout} = 0.0;
\]

\[
V(\text{out}) <+ \text{vout};
\]

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 of the form:

```
(terminal) branch_name
```

Examples

```
branch (b ,e)      base_emitter;
branch (b ,c)      base_collector;
```

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

```
Vin = V(in);       // Assigns Vin to the potential from net in to ground
Vdiode = V(p,n);   // Assigns Vdiode potential across net p and n
CurrentThruBranch = I(myBranch); // Assign current
```

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.

Once a value is indirectly assigned to a branch, it is illegal to then contribute to the branch.

**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,

\[
V(n1, n2) <+ \text{expression};
\]

\[
I(n1, n2) <+ \text{expression};
\]

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.

Contributing a flow to a branch that had been assigned a potential causes the branch to be converted to a flow branch, and the potential value is discarded. Contributing a voltage to a branch that was already assigned a flow will cause the flow value to be discarded and the branch to be converted to a flow source. This is described in the next section, switch branches.

### Switch Branches

A source branch can change between a potential and source flow. Assigning to a branch’s potential switches a branch into a potential source. Assigning to a branch’s flow converts the branch to a flow source. The last assignment determines whether the switch branch is a potential or flow source.

If a value is not assigned to a branch then the branch flow is automatically set to zero (0).
2. The switch position determines whether the circuit will behave like a potential or flow branch.

**Example**

In this example, an ideal relay is implemented using switch branches:

```verilog
module relay (p, n, switch_pos_p, switch_pos_n);
    electrical p, n, ps, ns;
    parameter Vth=1.0;
    analog begin
        if ((V(switch_pos_p, switch_pos_n) > Vth ? 1 : 0)
            V(p, n) <+ 0; // Potential source
        else
            I(p, n) <+ 0; // Flow source
    end
endmodule
```
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_primitive [ #( .param1( expr ) [, .param2( expr ) ] ) ] instance_name ( [node {, node} ]);
```

Note that if no ports are defined, the empty parentheses are still required.

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 vcol connected to out and
// lo with gain = 0.5, fc = 2k
vco #(0.5, 2000.0) vcol(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

// 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 vcol connected to out and
// lo with gain = loopGain/2, fc = fc
vco #(.gain(loopGain/2), .fc(fc) ) vcol(out, lo);

Paramsets

Paramsets provide a convenient way to collect parameter values for a particular
module. By doing so, the instance need only provide overrides for a smaller number
of parameters, similar to the SPICE model card concept. In the same manner as the
SPICE model card, paramsets may be overloaded.

A paramset can have a description attribute, which shall be used by the simulator
when generating help messages for the paramset.

Syntax

The paramset definition is enclosed between the paramset and endparamset
keywords. The first identifier is the name of the paramset, the second identifier is
the name of the module or other paramset with which this paramset is associated.

    paramset paramset_identifier module_or_paramset_identifier;
<paramset declarations>
<paramset statements>
endparamset

The statements contain no behavioral code (the restrictions are listed in the next section).

**Paramset statements**

The restrictions on statements or assignments allowed in paramsets are similar to those for analog functions. A paramset:

- can use any statements available for conditional execution
- cannot use access functions;
- cannot use contribution statements or event control statements; and
- cannot use named blocks.

The special syntax

```
.module_parameter_identifier = constant_expression ;
```

is used to assign values to the parameters of the associated module. The right-hand side expression can be composed of numbers, parameters, and hierarchical out-of-module references to parameters of a different module.

**Paramset overloading**

Paramset identifiers need not be unique: multiple paramsets can be declared using the same `paramset_identifier`, and they may refer to different modules. During elaboration, the simulator chooses an appropriate paramset from the set that shares a given name for every instance that references that name.

When choosing an appropriate paramset, the simulator uses the following rules:

- All parameters overridden on the instance must be parameters of the paramset
- The parameters of the paramset, with overrides and defaults, must be all within the allowed ranges specified in the paramset parameter declaration.
- The local parameters of the paramset, computed from parameters, must be within the allowed ranges specified in the paramset.
If the rules above are not sufficient for the simulator to pick a unique paramset, then the simulator applies the following rules in order until a unique paramset has been selected:

- The paramset with the fewest number of un-overridden parameters shall be selected.

- The paramset with the greatest number of local parameters with specified ranges shall be selected.

An error will occur if there are still more than one applicable paramset for an instance after application of these rules.

An error will occur if a paramset assigns a value to a module parameter and this value is outside the range specified for that module parameter. The simulator considers only the ranges of the paramset’s own parameters when choosing a paramset.

**Paramset output variables**

As with modules, integer or real variables in the paramset that are declared with descriptions are considered output variables and are available for print out or storage by the simulator. Certain rules apply to paramset output variables and output variables of modules referenced by a paramset:

- If a paramset output variable has the same name as an output variable of the module, the value of the paramset output variable is the value that is reported for any instance that uses the paramset.

- If a paramset variable without a description has the same name as an output variable of the module, the module output variable of that name will not be available for instances that use the paramset.

- A paramset output variable’s value can be computed from values of any output parameters of the module by using the special syntax (note period (.)):

  
  `module_output_variable_identifier`

**Example**

The paramset `my_paramset` can be instantiated in the netlist rather than the module `my_simple_module`. 

paramset my_paramset my_simple_module;
parameter real real_param1 = 1.0;
.outvalue = real_param1;
endparamset

module my_simple_module(out);
electrical out;
parameter real outvalue = 0.0;
analog
  V(out) <+ outvalue;
endmodule

Ports Assignments

Ports are used to interconnect instances of modules. The port type and direction are declared within the module body.

Port Assignment

The port type is declared by giving it a discipline. Ports that do not have a type declared cannot be accessed in a behavioral description and can only be used in a structural description where they are passed to instances of modules.

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

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
...
Hierarchical system parameters

There are six system parameters that are implicitly declared for every module:

$\text{mfactor}$, $\text{xposition}$, $\text{yposition}$, $\text{angle}$, $\text{hflip}$, and $\text{vflip}$.

The values of these parameters may be accessed in a module or paramset using these names in the usual manner. The value of these parameters may be overridden module instance parameter value assignment by name, by a paramset, or by using the \texttt{defparam} statement. The system parameter identifier is prefixed by a period (.), just as for explicitly-declared parameters.

The simulator automatically manages the proper multiplication of contributions based on the instances multiplication factor, so it is not expected that the user will modify the code based on the value of the $\text{mfactor}$. If an instance of a module has a value of $\text{mfactor}$ other than one (1), then the following rules are applied automatically by the simulator, and the simulator will warn if it detects possible double scaling:

- All contributions to a branch flow quantity in the analog block are multiplied by $\text{mfactor}$
- The value returned by any branch flow probe in the analog block, including those used in indirect assignments, are divided by $\text{mfactor}$
- Contributions to a branch flow quantity using the noise functions are multiplied by $\text{mfactor}$
- Contributions to a branch potential quantity using the noise functions are divided by $\text{mfactor}$
- The module’s value of $\text{mfactor}$ is also propagated to any module instantiated by the original module, according to the hierarchical rules.

The values of the parameters can be accessed in the same manner as other system parameters using the hierarchical system parameter functions.
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.

Example

```
analog begin : the_module
    x = 1.0;
    if (x == 1) begin : the_local_block
        real x;
        x = 0; // This assignment has no effect on output
    end
    V(out) <+ x; // x = 1.0
end
```

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;
    output_declaration;
```
statement_block;
endfunction

The *input_declaration* describes the input parameters to the function:

```verilog
input passed_parameters;
real parameter_list;
```

The *output_declaration* describes the output parameters returned by the function:

```verilog
input passed_parameters;
real parameter_list;
```

The *inout_declaration* describes parameters that are both passed into and passed back by the function.

```verilog
inout 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. Output and inout variables can be read and assigned within the flow of the analog block; the last value assigned during function evaluation is the value returned by the function.

**Example**

```verilog
analog function real B_of_T;
    input B, T, T_NOM, XTB;
    real B, T, T_NOM, XTB;
    begin
```
\begin{verbatim}
B_of_T = B * pow(T / T_NOM, XTB);
end
endfunction

The function is called by the line,

\texttt{BF_T = B_of_T(BF, T, T_NOM, XTB);}

The analog function \texttt{mult} takes three inputs and returns a polynomial result as well as the intermediate terms.

\begin{verbatim}
analog function poly;
inout a1, a2, a3;
real a1, a2, a3;
begin
    a1 = a1;
    a2 = a2*a2;
    a3 = a3*a3*a3;
    poly = a1 + a2 + a3;
end
endfunction
\end{verbatim}
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
- comment
- operator
- number
- string
- identifier
- keyword

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.

White Space

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

Comments

There are two ways to include comments:

- A single line comment starts with // and continues to the end of the line.
- Block comments which begin with /* and end with */

Example
// This is a single line comment

/* This is a block comment which can include any ASCII character */

Block statements cannot be nested but can include single line comments.

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
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.

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>u</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
110000

Keywords

Keywords are predefined non-escaped identifiers. Keywords define the language constructs. They are defined in lowercase only. Appendix 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.

Examples

deviceName
i
Vth0
vth0
_device
sheet_rho$

Escaped Identifiers

Escaped identifiers begin with the backslash character (\) and end with white space (either a space, tab, newline, or formfeed). This allows identifiers to use any printable ASCII characters (the decimal values 33 through 126 or 21 through 7E in hexadecimal).

The leading back-slash character and the terminating white space are not considered to be part of the identifier. Therefore, an escaped identifier \bus123 is treated the same as a non-escaped identifier bus123.
Examples
\bus+index
\-clocktick
\net1/\net2
\{x,y\}

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.

Examples
`ifdef NOISE
`endif

Attributes

Attributes provide a general mechanism for specifying properties about objects, statements, and groups of statements in the Verilog-A source that may be used by the simulator in various ways to control the operation or behavior of the tool.

Syntax
(* attr_spec {,attr_spec } *)

where
    attr_spec {= constant_expression}

Example
(* desc = "This is the an output variable" *) real myOutputVar;

An attribute can appear in the Verilog-A description as a prefix attached to certain module item declarations. If a value is not specifically assigned to the attribute, then its value is 1. If the same attribute name is defined more than once for the same language element, the last attribute value is used and a warning that a duplicate attribute specification has occurred will be issued.
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;
```

Output variables

The standard attributes for descriptions and units can have a special meaning for variables declared at module scope. Module scope variables with a description or units attribute, or both, are called output variables, and can be output through the usual simulator output mechanism.

Example

A module for a transistor with the following declaration at the module scope provides the output variable $g_m$. 
(* desc="Gate Transconductance", units="S" *) real gm;

The simulator would output the information in a .print statement as:

\[
 gm = 5.12\times10^{-3} \text{ S Gate Transconductance}
\]

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

**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,

\[
 I(d_i,s_i) <+ \text{white_noise}(4 \times \text{\'P_K} \times T \times (2/3) \times \text{abs}(gm),
                        "shot");
\]

will always return 0 since the $2/3$ term 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.

**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.

```markdown
// Electrical
```
// Current in amperes

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

// Charge in coulombs

```verilog
nature Charge
units = "coul";
access = Q;
ddt_nature = Current;
`ifdef CHARGE_ABSTOL
  abstol = `CHARGE_ABSTOL;
`else
  abstol = 1e-14;
`endif
endnature
```

// Potential in volts

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

### Ground Declaration

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

**Syntax**

```verilog
ground list_of_nets;
```

where `list_of_nets` is a comma-separated list of nets.
Example
```
`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
```
`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);
  blk_c c1 (Output1, Output2, Output3, c_b1, c_b2);
endmodule
```

Genvar

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

Syntax
```
genvar list_of_genvar_identifiers;
```

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

Example
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 initialization is via a list of constant expressions containing only constant numbers and previously defined parameters within bracket delimiters, {}.

Parameter values cannot be modified at runtime.

Syntax

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

where the `list_of_assignments` is a comma separated list of

```
parameter_identifier = constant [value-range]
```

where `value-range` is of the form

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

where the `value_rangeSpecifier` is of the form

```
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. Value ranges can have simple exclusions as well.

String Parameters
Strings are useful for parameters that act as flags or other text input. The set of allowed values for the string can be specified as a comma-separated list of strings inside curly braces, but ranges (and exclusions) are not allowed.

The operators that can be used with string parameters are restricted to the two listed in the table below.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equality. Checks if the two strings are equal. Result is 1 if they are equal and 0 if not. Both operands can be string parameters, or one can be a string parameter and the other a constant string literal.</td>
</tr>
<tr>
<td>!=</td>
<td>Inequality.</td>
</tr>
</tbody>
</table>

Table: Allowed string operations.

Examples

/* Define a parameter of type real with a default value of 0 and allowed values between 0 and up to, but not including, infinity and excluding values between 10 and 100 (however, 10 and 100 are acceptable) and 200 and 400 (200 is acceptable, but 400 is not.) */

```verilog
parameter real TestFlag = 0 from [0:inf) exclude (10:100)
exclude (200:400);
```

/* Define a real parameter with a default value of 27, ranging from -273.15 up to, but not including infinity. */

```verilog
parameter real Temp = 27 from [-273.15:inf);
```

/* Define a parameter R with a default value of 50, ranging from, but not including, 0 to infinity. R is implicitly defined as type integer. */

```verilog
parameter R = 50 from (0:inf);
```

/* Define a string parameter Answer with only 2 allowed values: YES or NO. */
### Local Parameters

Local parameters are identified by the `localparam` keyword. They are identical to parameters except that they cannot be modified *directly* with the `defparam` statement or by the ordered or named parameter value method in hierarchical assignments. However, local parameters can be assigned to a constant expression containing a parameter that is modified with the `defparam` statement or by the ordered or named parameter value assignment.

### Examples

```verilog
parameter real my_reg_param = 1;
localparam real my_local_param = my_reg_param;
```

### Parameter Aliases

Parameters can have aliases to allow an alternate name to be used when overriding module parameters. For example, parameters with different names may be used for the same purpose to support conventions in different simulators.

Parameter aliases follow these rules:

- The type of an alias (`real`, `integer`, or `string`) is determined by the original parameter, as well as its range of allowed values (if specified).
- The `alias_identifier` cannot occur anywhere else in the module.
- Equations in the module must reference the parameter by its original name (not the alias).
- Multiple aliases can point to the same parameter.
- It is not valid to specify an override for parameter by its original name and one or more aliases, or by more than one alias, regardless of how the override is done (i.e., by name or using the `defparam` statement).
- When the simulator generates a list of parameter values used (e.g., for an operating point analysis) the original name will appear in the list.

### Examples
/* Define parameters */
parameter real real_param = 1.1 from [0:10];
parameter integer integer_param = 1 from [0:10];

/* Create various aliases */
aliasparam real_alias=real_param;
aliasparam mult_real_alias=real_param;
aliasparam integer_alias=integer_param;
aliasparam mult_integer_alias=integer_param;
Chapter 5: Analog Block Statements

This chapter describes the analog 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.

Sequential Block

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

Syntax

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

The optional block identifier allows for naming of the block. Named blocks allow local variable declaration.

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.

Syntax

```verilog
if ( expression ) true_statement;
[ else false_statement; ]
```
If the expression evaluates to true (non-zero), then the true_statement will be executed. 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.

Example

```verbatim
if (Vd < 0) begin
  if (Vd < -Bv)
    Id = -Area * Is_temp * \( \text{limexp}(-\frac{Bv + Vd}{Vth}) + \frac{Bv}{Vth} \);
  else if (Vd == -Bv)
    Id = -Area * Ibv_calc;
  else if (Vd <= -5 * N * Vth)
    Id = -Area * Is_temp;
  else // -5 \frac{nKT}{q} <= Vd < 0
    Id = Area * Is_temp * (\text{limexp}(Vd / Vth) - 1);
end
else
  Id = Area * Is_temp * (\text{limexp}(Vd / (N * Vth)) - 1);
```

Case Statement

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

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

where case_item is:

```verbatim
  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

```verilog
    case(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 RGEO = %d not matched
      (BSIM4RdsEndIso)\n", rgeo);
    endcase
```

Repeat Statement

The `repeat()` statement executes a statement a fixed number of times. The number is given by the `repeat` expression.

Syntax

```
repeat (expression) statement
```

Example

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

While Statement

`while()` executes a statement until its control expression becomes `false`. If the expression is `false` when the loop is entered, the statement is not executed at all.

Syntax
while (expression) statement

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 only. No use of procedural assignments and expressions are allowed.

Syntax

    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.

1. Unary/Binary/Ternary Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ - */</td>
<td>Arithmetic</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
<tr>
<td>&gt; &gt;= &lt; &lt;=</td>
<td>Relational</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></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>

Arithmetic Operators

The arithmetic operators are summarized in table 2.
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 table 10 for precedence relations.

Relational Operators

Table defines and summarizes the relational operators.

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

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

Logical Operators

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

<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.

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>

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>

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>

8. Bitwise exclusive nor operator

Verilog-A Reference Manual
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

```verilog
target
  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).

Syntax

```verilog
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

BSIMvth0 = (BSIM3type == `NMOS) ? 0.7 : -0.7;

**Precedence**

Table 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.

10. **Precedence**

<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

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

The expression evaluation follows the order precedence described in 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*.

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, then 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 11.

11. 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>
</tbody>
</table>
Verilog-A Reference Manual

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>log(x)</td>
<td>log base 10</td>
<td>x&gt;0</td>
</tr>
<tr>
<td>exp(x)</td>
<td>exponential</td>
<td>x&lt;80</td>
</tr>
<tr>
<td>sqrt(x)</td>
<td>square root</td>
<td>x&gt;=0</td>
</tr>
<tr>
<td>min(x,y)</td>
<td>minimum of x and y</td>
<td>all x, y</td>
</tr>
<tr>
<td>max(x,y)</td>
<td>maximum of x and y</td>
<td>all x, y</td>
</tr>
<tr>
<td>abs(x)</td>
<td>absolute value</td>
<td>all 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>
</tr>
<tr>
<td>floor(x)</td>
<td>floor</td>
<td>all x</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>Ceiling</td>
<td>all x</td>
</tr>
</tbody>
</table>

For the `min()`, `max()`, and `abs()` functions, the derivative behavior is defined as:

- `min(x,y)` is equivalent to `(x < y) ? x : y`
- `max(x,y)` is equivalent to `(x > y) ? x : y`
- `abs(x)` is equivalent to `(x > 0) ? x : -x`

Transcendental Functions

The transcendental functions supported by Verilog-A are shown in table 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.

12. 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 (pi/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>Function</td>
<td>Description</td>
<td>Domain</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------</td>
<td>----------</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>atanh(x)</td>
<td>arch-hyperbolic tangent</td>
<td>$-1 &lt;= x &lt;= 1$</td>
</tr>
</tbody>
</table>

### 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**

```verilog
integer seed_value, random_value;
random_value = $random;
// returns a value between -31 and 31.
random_value = $random(seed_value) % 32;
```
The $dist_uniform and $rdist_uniform Functions

The $dist_uniform() and $rdist_uniform() functions return uniform distributions across the range. Use $dist_uniform() to return integer values and $rdist_uniform() to return real values.

Syntax

```
$dist_uniform( seed_expression, start_expression, end_expression );
$rdist_uniform( seed_expression, start_expression, end_expression );
```

where

- the start and end real parameters bound the values returned. The start value must be smaller than the end value.
- The $dist_uniform() parameters start_expression and end_expression are integer values, and for $rdist_uniform(), are real values.

Example

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

The $dist_normal and $rdist_normal Functions

The $dist_normal() and $rdist_normal() functions return normal distributions around a mean value. Use $dist_normal() to return integer values and $rdist_normal() to return real values.

Syntax

```
$dist_normal( seed_expression, mean_expression, stdev_expression );
$rdist_normal( seed_expression, mean_expression, stdev_expression );
```

where

- stdev_expression determines the shape (standard deviation) of the density function. It is an integer value for $dist_normal and a real value for $rdist_normal.
• 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. It is an integer value for $dist_normal and a real value for $rdist_normal.

• The mean_expression parameter causes the average value of the return value to approach the mean_expression.

Example

// Returns a Guassian distribution
random_value = $rdist_normal(mySeed, 0.0, 1.0);

The $dist_exponential and $rdist_exponential Functions

The $dist_exponential() and $rdist_exponential() functions generate a distribution that follows an exponential. Use $dist_exponential() to return integer values and $rdist_exponential() to return real values.

Syntax

$dist_exponential( seed_expression, mean_expression );

$rdist_exponential( seed_expression, mean_expression );

where

• mean_expression parameter causes the average value of the return value to approach the mean. The mean_expression value must be greater than zero (0). It is an integer value for $dist_exponential and a real value for $rdist_exponential.

Example

// Exponential distribution approaching 1
random_value = $rdist_exponential(mySeed, 1);

The $dist_poisson and $rdist_poisson Functions

The $dist_poisson() and $rdist_poisson() functions return a Poisson distribution centered around a mean value. Use $dist_poisson() to return integer values and $rdist_poisson() to return real values.
Syntax

$\text{dist\_poisson}( \text{seed\_expression}, \text{mean\_expression} );$

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

where

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

- The \text{mean\_expression} parameter causes the average value of the return value to approach the \text{mean\_expression}. It is an integer value for $\text{dist\_poisson}$ and a real value for $\text{rdist\_poisson}$.

Example

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

The $\text{dist\_chi\_square}$ and $\text{rdist\_chi\_square}$ Functions

The $\text{dist\_chi\_square}()$ and $\text{rdist\_chi\_square}()$ functions returns a Chi-Square distribution. Use $\text{dist\_chi\_square}()$ to return integer values and $\text{rdist\_chi\_square}()$ to return real values.

Syntax

$\text{dist\_chi\_square}( \text{seed\_expression}, \text{degree\_of\_freedom\_expression} );$

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

where

- \text{degree\_of\_freedom\_expression} parameter helps determine the shape of the density function. Larger values spread the returned values over a wider range. The \text{degree\_of\_freedom\_expression} value must be greater than zero (0). It is an integer value for $\text{dist\_chi\_square}$ and a real value for $\text{rdist\_chi\_square}$.

Example

// Chi-Square
random\_value = $\text{rdist\_chi\_square}(\text{mySeed}, 1.0);$
The $dist_t and $rdist_t Functions

The $dist_t() and $rdist_t() functions returns a Student’s T distribution of values. Use $dist_t() to return integer values and $rdist_t() to return real values.

Syntax

$dist_t( seed_expression, degree_of_freedom_expression );
$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 must be greater than zero (0). It is an integer value for $dist_t and a real value for $rdist_t.

Example

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

The $dist_erlang and $rdist_erlang Functions

The $dist_erlang() and $rdist_erlang() functions return values that form an Erlang random distribution. Use $dist_erlang() to return integer values and $rdist_erlang() to return real values.

Syntax

$dist_erlang( seed_expression, k_stage_expression, mean_expression );
$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. It is an integer value for $dist_erlang and a real value for $rdist_erlang.
Example

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

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 an iterative approach 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.

NOTE: Tolerance arguments are not supported for all operators.
Parameters

Some analog operators (Laplace and Z-transform filters) require arrays as arguments. A literal array can be passed into the function using the brace operators.

Examples

```verilog
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

```verilog
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**

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

**Circular Integrator Operator**

The circular integrator operator, *idtmod()* , converts an expression argument into its indefinitely integrated form.

**Syntax**

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

where

• *expr* is the expression to be integrated.

• *ic* is an optional expression specifying an initial condition. The default value is zero (0).

• *modulus* is a positive-valued expression which specifies the value at which the output of *idtmod()* is reset. If not specified, *idtmod()* behaves like the *idt()* operator and performs no limiting on the output of the integrator.

• *offset* is a dynamic value added to the integration. The default of *offset* is zero (0).

• *abstol* is a constant absolute tolerance to be applied to the input of the *idtmod()* operator and defines the largest signal level that can be considered to be negligible.

The *modulus* and *offset* parameters define the bounds of the integral. The output of the *idtmod()* function always remains in the range:
offset <= idtmod_output < offset + modulus

Example

phase = idtmod(fc + gain * V(in), 0, 1, 0);

Derivative Operator

The derivative operator, \texttt{ddx()}, provides access to the symbolically-computed partial derivative of its argument with respect to a state variable.

Syntax

\[
\texttt{ddx( expr, potential_or_flow(name) )}
\]

where \texttt{expr} is an expression with respect to which the derivative will be taken.

Examples

\[
\texttt{dxdv = ddx(x, V(in));}
\]
\[
\texttt{x1 = ddx(ddx(V(in1)*\sin(V(in2)), V(in1)), V(in2));}
\]

The operator returns the partial derivative of its first argument with respect to the second argument. It holds all other unknowns fixed and evaluates the expression at the current operating point. The second argument must be the potential of a scalar net or port or the flow through a branch (the unknown variables, typically voltages and currents, in the system of equations for the analog solver).

If the expression does not depend explicitly on the second argument, then the \texttt{ddx()} operator returns zero (0).

Absolute Delay Operator

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

Syntax

\[
\texttt{absdelay( expr, time_delay [, max_delay] )}
\]

where
• \textit{expr} is the expression to be delayed

• \textit{time\_delay} is a nonnegative expression that defines how much \textit{expr} is to be delayed

• If the optional \textit{max\_delay} is specified, the value of \textit{time\_delay} can change during a simulation, as long as it remains positive and less than \textit{max\_delay}. If \textit{max\_delay} is not specified, any changes to \textit{time\_delay} are ignored. If \textit{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, \texttt{absdelay()} returns the value of \textit{expr}. In AC and small-signal analyses, the input waveform \textit{expr} is phase shifted according to:

In the time domain, \texttt{absdelay()} introduces a delay to the instantaneous value of \textit{expr} according to the formula:

\begin{align*}
\text{Example} \\
V_{\text{delayed}} = \texttt{absdelay}( V(in), \textit{time\_delay} )
\end{align*}

\section*{Transition Filter}

The transition filter, \texttt{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, \texttt{slew()}.

\subsection*{Syntax}

\begin{verbatim}
transition ( expr [ , time\_delay [ , rise\_time [ , fall\_time [ , time\_tol ]] ]] )
\end{verbatim}

where

• all values are real and \textit{time\_delay}, \textit{rise\_time}, \textit{fall\_time}, and \textit{time\_tol} are optional

• \textit{expr} is the input expression waveform to be delayed

• \textit{time\_delay} is the delay time and must be \(\geq 0\) (defaults to zero (0))

• \textit{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.

The figure below shows an example of a `transition()` filter on a pulse waveform.

3. Transition Filter on Pulse Waveform.

If interrupted on a rising transition, the function will attempt to finish the transition in the specified time with the following rules:

- 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.
4. The Transition Function Completion after Interruption

**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
  `(max_neg_slew_rate` is optional and must be < 0; If not specified, it defaults to the negative of `max_pos_slew_rate`)

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, \texttt{slew()} returns \texttt{expr} unchanged. If the slope of \texttt{expr} is in-between the maximum slew rates, the input \texttt{expr} is returned.

In DC analyses, the input \texttt{expr} is passed through the filter unchanged. In AC small-signal analyses, the \texttt{slew()} operator has a unity transfer function. In this case it has zero transmission.

![Diagram of signal with slopes](image)

5. Slew Rate Limiting of Slope.

## 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}
\texttt{last_crossing( expr, dir )}
\end{verbatim}

where

- \textit{expr} is the signal expression
- \textit{dir} is an integer flag with values -1, 0, +1

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

If \textit{expr} has not crossed zero, the function will return a negative value.
Limited Exponential

An alternative function to the \( \exp() \) standard mathematical function is the \( \limexp() \) function. The \( \limexp() \) function is mathematically equivalent to the \( \exp() \) 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 \( \limexp() \) equals the exponential \( \exp() \) for that input.

Syntax

\[
\limexp( \text{arg} );
\]

Example

\[
I_s = I_{s0} \times \limexp(V_j / \$vt);
\]

Laplace Transform Filters

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

\( \text{laplace_zp()} \)

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

Syntax

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

where

- \( \text{expr} \) 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:

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

where \( \zeta \) and \( \rho \) are the real and imaginary parts of the \( k \)th zero and are the real and imaginary parts of the \( k \)th pole. For a real pole or real 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**

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

where \( \zeta \) and \( \rho \) 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.

**laplace_np()**

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

**Syntax**

```markdown
laplace_np(expr, n, \rho)
```

where

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

The transfer function is:

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

where \( \zeta \) and \( \rho \) 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  

```verilog
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:

$$H(s) = \sum_{k=0}^{M} n_{k} s^{k} \sum_{k=0}^{N} d_{k} s^{k}$$

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 $t0$.

- $T$ specifies the period of the filter, is mandatory, and must be positive.
- $t$ specifies the transition time, is optional, and must be non-negative.
• \( t0 \) specifies the first transition time. If it is not supplied, the first transition is at \( t=0 \).

### zi_zp() 

The \( zi\_zp() \) operator implements the zero-pole form of the Z-transform filter.

#### Syntax

\[
zi\_zp( expr, \zeta, \rho, T[, \tau[, t0]] )
\]

where

- \( expr \) is the expression to be transformed.
- \( \zeta \) (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.
- \( \rho \) (rho) is a vector of \( N \) real pairs, one for each pole, represented in the same manner as the zeros.

The transfer function is:

\[
H(z) = \frac{\prod_{k=0}^{M-1} (1 - z^{-1}(\zeta'_k + j\zeta'_i))}{\prod_{k=0}^{N-1} (1 - z^{-1}(\rho'_k + j\rho'_i))}
\]

where and are the real and imaginary parts of the \( k^{th} \) zero, while and are the real and imaginary parts of the \( k^{th} \) 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

\[ zi_{zd}( \text{expr}, \zeta, d, T [, \tau [, t_0 ]] ) \]

where

- \text{expr} is the expression to be transformed.
- \zeta (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:

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

where \( \zeta_k^r \) 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.

\text{zi_np()}

The \text{zi_np()} implements the numerator-pole form of the Z-transform filter.

Syntax

\[ zi_{np}( \text{expr}, n, \rho, T [, \tau [, t_0 ]] ) \]

where

- \text{expr} is the expression to be transformed.
- \( n \) is a vector of \( M \) real numbers containing the coefficients of the numerator.
• \( \rho \) (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:

\[
H(z) = \frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\prod_{k=0}^{N-1} (1 - z^{-1}(\rho^r_k + j\rho^i_k))}
\]

where \( n_k \) is the coefficient of the \( k^{th} \) power of \( z \) in the numerator, while \( \rho^r_k \) and \( \rho^i_k \) 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**

```plaintext
zi\_nd( expr, n, d, T [, \tau [ [, t0 ]]] )
```

where

- `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:

\[
H(z) = \frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\sum_{k=0}^{N-1} d_k z^{-k}}
\]
where $n_k$ is the coefficient of the $k^{th}$ power of $s$ in the numerator and $d_k$ is the coefficient of the $k^{th}$ power of $z$ 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

```verilog
@(initial_step("tran","ac","dc"))
```

The final_step Event

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

Syntax

```verilog
@( 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

```verilog
@(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 13 defines the return code for the particular event and analysis type.

13. Return Codes for intial_step and final_step

<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 1</td>
<td>0 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 1</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 1</td>
<td>0 0 0</td>
</tr>
<tr>
<td>final_step()</td>
<td>0</td>
<td>0 0 1</td>
<td>0 0 1</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 1</td>
<td>0 0 1</td>
</tr>
</tbody>
</table>
**Monitored Events**

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

**The cross Function**

The cross function, `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 `cross()` function can control the timestep to accurately resolve the crossing.

**Syntax**

```plaintext
cross( expr [, dir [, time_tol [, expr_tol ]]] ]);
```

where

- `expr` is a required argument
- `dir` is an optional argument that is an integer expression
- `time_tol` and `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 `dir` is set to 0 or is not specified, the event and timestep control will occur on both positive and negative signal crossings. If `dir` is +1, then the event and timestep control occurs on the rising transitions of the signal only. If `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 `cross()` function will not generate an event. The `expr_tol` and `time_tol` arguments represent the maximum allowable error between the estimated crossing point and the actual crossing point.
Example

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

\begin{verbatim}
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
\end{verbatim}

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

Above Function

The above function provides a way to generate an event when a specified expression becomes greater than or equal to zero. An above event can be generated and detected during initialization as compared to a cross event, which can be generated and detected only after at least one transient time step is finished.

Syntax

\begin{verbatim}
above (expr [ , time_tol [ , expr_tol ] ] )
\end{verbatim}

where

\texttt{expr} is a real expression whose value is to be compared to zero.

\texttt{time_tol} is a constant real, positive expression and is the largest non-negligible time interval.

\texttt{expr_tol} is a constant real, positive expression and is the largest non-negligible difference.
If specify expr_tol is specified both it and time_tol must be satisfied. If expr_tol is not specified, the simulator uses the value of its own reltol parameter.

During a transient analysis, after time t = 0, the above function then behaves the same as a cross function specified as

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

During a transient analysis, the time steps are controlled by the above function to accurately resolve the time when expr rises to zero or above. The above function is subject to the same restrictions as other analog operators. That is, it can not be used inside if, case or for statements unless these statements are controlled by genvar-constant expressions.

**Example**

The following example illustrates to use the above function. The function generates an above event each time the analog voltage increases through the value specified by $V_{th\_HI}$ or decreases through the value specified by $V_{th\_LO}$.

```verilog`
`include "disciplines.vams"
`define HIGH 1
`define LOW 0

module test_above(in, out);
electrical in, out;
parameter real Vth_HI = 3;
parameter real Vth_LO = 1;
real out_value;
analog begin
  @(above(V(in) - Vth_HI))
    out_value = `HIGH;
  @(above(Vth_LO - V(in)))
    out_value = `LOW;
  V(out) <+ out_value;
end
endmodule
```

**The timer Function**

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

```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

```verilog
@(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 in the statement block cannot have expressions that include analog operators.
- The statements in the statement block cannot be contribution statements.
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, $\text{realtime}$, returns the simulation time in seconds. It is a deprecated function and $\text{abstime}$ should be used instead.

Syntax

$\text{realtime}()$

Example

CurTimeIn_mS = $\text{realtime}();$

The Thermal Voltage Function

The thermal voltage function, $\text{vt}$, 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

$\text{vt}([\text{temperature}\_\text{expression}])$

Example

DevVth = $\text{vt}(\text{Tnom} + '\text{P}\_\text{CELSIUS0}); // \text{Tnom in C}$

Note: The macro P\_CELSIUS0, defined in the constants.vams header file, provides a convenient way to offset temperatures.

The Simulator Parameter Function

The simulator parameter function, $\text{simparam}()$, queries the simulator for a simulation parameter named by the string argument that it is passed. If the string is known to the simulator, its value is returned. If string is not known to the simulator an optional expression is returned, or if the optional expression is not supplied as an argument, then an error is generated.

$\text{simparam}()$ returns a real value even the internal simulator value is an integer.
There is no fixed list of simulation parameters. However, simulators at the minimum accept the strings in the table below.

<table>
<thead>
<tr>
<th>String</th>
<th>Units</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gdev</td>
<td>1/Ohms</td>
<td>Additional conductance to be added to nonlinear branches for conductance homotopy convergence algorithm</td>
</tr>
<tr>
<td>gmin</td>
<td>1/Ohms</td>
<td>Minimum conductance placed in parallel with nonlinear branches</td>
</tr>
<tr>
<td>imax</td>
<td>Amps</td>
<td>Branch current threshold above which the constitutive relation of a nonlinear branch should be linearized</td>
</tr>
<tr>
<td>imelt</td>
<td>Amps</td>
<td>Branch current threshold indicating device failure</td>
</tr>
<tr>
<td>iteration</td>
<td></td>
<td>Iteration number of the analog solver</td>
</tr>
<tr>
<td>scale</td>
<td></td>
<td>Scale factor for device instance geometry parameters</td>
</tr>
<tr>
<td>shrink</td>
<td></td>
<td>Optical linear shrink factor</td>
</tr>
<tr>
<td>simulatorSubversion</td>
<td></td>
<td>The simulator sub-version</td>
</tr>
<tr>
<td>simulatorVersion</td>
<td></td>
<td>The simulator version</td>
</tr>
<tr>
<td>sourceScaleFactor</td>
<td></td>
<td>Multiplicative factor for independent sources for source stepping homotopy convergence algorithm</td>
</tr>
<tr>
<td>tnom</td>
<td>degrees Celsius</td>
<td>Default value of temperature at which model parameters were extracted</td>
</tr>
</tbody>
</table>

**Example**
In this example, the variable `myTnomK` is set to the simulator’s parameter named `tnom`, if it exists, otherwise, the value 27.0 degrees Celsius is returned (and offset to Kelvin).

```verilog
myTnomK = $simparam("tnom", 27.0) + `P_CELSIUS0;
```

## 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);
```

### Limiting Functions

The `$limit` function provides a way to indicate that a particular function has a strong nonlinearity and to provide suggested ways to limit the change at each solver iteration.

**Syntax**

```
$limit(access_function_reference [, analog_function_identifier [, arg_list ]])
```
When the simulator has converged, the return value of the \$limit() function is the value of the access function reference (within appropriate tolerances). The simulator determines if limiting should be applied and what the return value is on a given iteration. When more than one argument is supplied to the \$limit() function, the second argument recommends a function to use to compute the return value. When the second argument is a string, it refers to a built-in function of the simulator, typically, \textit{pnjlim} and \textit{fetlim}. If the string refers to an unknown or unsupported function, the simulator will try to apply the appropriate limiting algorithm, just as if no string had been supplied.

If the function \textit{pnjlim} is used, two additional arguments to the \$limit() function are required when the third argument to \$limit() indicates a step size \textit{vte} and the fourth argument is a critical voltage \textit{vcrit}. The step size \textit{vte} is usually defined as the product of the thermal voltage \$vt and the emission coefficient of the junction. The critical voltage is generally determined from

\[
V_{crit} = vte \times \ln(vte (\sqrt{2} \times Is))
\]

where \textit{Is} is the saturation current of the junction.

The \textit{fetlim} limiting function is intended to help limit the potential across the oxide of a MOS transistor. The third argument to the \$limit() function should be the threshold voltage of the MOS transistor. The second argument of the \$limit() function may be an identifier referring to a user-defined analog function. When limiting is needed (as determined by the simulator), it will pass the following quantities as arguments to the user-defined function (all the arguments of the user-defined function must be declared as input):

- The first argument of the user-defined function is the value of the access function reference for the current iteration.
- The second argument is the appropriate internal state; generally, this is the value that was returned by the \$limit() function on the previous iteration.
- If more than two arguments are passed to the \$limit() function, then the third and subsequent arguments are passed as the third and subsequent arguments of the user-defined function.

Example

The following is a simple (contrived) user-defined function for limiting a voltage.

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

    analog function real mylimiter;
```
input new, old, x, y;
real new, old, x, y;
    mylimiter = x + y + new + old / 10;
endfunction

analog begin
    I1 = I(p, n);
    I(p, n) <+ $limit(V(p, n), mylimiter, I1*50, 1);
end
endmodule

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 [(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

@(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.

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, table 14. If a type is unknown, the simulator returns no match. The return codes for analysis functions are summarized in table 15.

14. 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>
15. Analysis Function Return Codes

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Argument</th>
<th>DC</th>
<th>TRAN</th>
<th>AC</th>
<th>NOISE</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>
</tr>
<tr>
<td>Initial DC state</td>
<td>&quot;static&quot;</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Initial condition</td>
<td>&quot;ic&quot;</td>
<td>0</td>
<td>1</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>
</tr>
<tr>
<td>Transient</td>
<td>&quot;tran&quot;</td>
<td>0</td>
<td>1</td>
<td>1</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>
</tr>
<tr>
<td>Noise</td>
<td>&quot;noise&quot;</td>
<td>0</td>
<td>0</td>
<td>0</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

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

```verbatim
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

```verbatim
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. It is used to generate pink noise with a power density that varies proportionally to the inverse of frequency.

Syntax

```verbatim
flicker_noise( pwr, exp [, name ] )
```

where

- the function generates a frequency-dependent noise of power `pwr` at 1 Hz which varies in proportion to the expression 1/f^`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

```verbatim
I(n1, n2) <+ flicker_noise(KF * pow(abs(I(n1,n2)), AF), 1.0, "flicker");
```
Noise Table Function

The noise table function, `noise_table()`, provides a means to introduce noise via a piecewise linear function of frequency.

Syntax

```plaintext
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 `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

```plaintext
I(n1, n2) <+ noise_table({1, 0.1, 100, 0.2, 1e5, 0.24}, "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.

Parameter given ($param_given) function

The $param_given() function can be used to test whether a parameter value was obtained from the default value in its declaration statement or if that value was overridden by a value passed from the netlist. The $param_given() function takes a single, required argument, the parameter identifier. The return value is one (1) if the parameter was overridden by a module instance parameter value assignment and zero (0) otherwise.

Syntax

$param_given( param_identifier )

where param_identifier is the name of a module parameter.

Example

In this example, if the netlist sets the value of vth0, then the variable BSIM3vth0 is set to this value. Otherwise the BSIM3vth0 is set to either 0.7 or -0.7 (depending on the value of BSIM3type).

if ($param_given(vth0)
   BSIM3vth0 = vth0;
else
   BSIM3vth0 = (BSIM3type == `NMOS) ? 0.7 : -0.7;

Interpolation (table model) Function

The interpolation function, $table_model(), allows the module to approximate the behavior of a system by interpolating between user-supplied data points. The user provides a dataset of points (x_1, x_2, .., x_N, y_i) such that f(x_1, x_2, .., x_N) = y_i, where f is the model function and N is the number of independent variables of the
model. These data points are stored in a text file and are accessed during the analysis by the Verilog-A module.

The interpolation algorithm then approximates the true model behavior at any point in the domain of the sampled data. Data points outside of the sampled domain will be approximated via extrapolation of the data within the domain. Extrapolated data can be inaccurate and should be avoided.

The Verilog-A algorithm is a piecewise-linear interpolation for the \$table_model() function. However, higher-order interpolation algorithms may be provided in a future revision of the language.

The \$table_model() system function has the same restrictions as analog operators. That is, it cannot be used inside of if(), case(), or for() statements unless these statements are controlled by genvar-constant expressions.

Syntax

\$table_model( table_inputs, table_data_source, table_control_string );

where

\textit{table_inputs} is an (optionally multi-dimensional) expression. For more information on the \textit{table_inputs} argument, see the Table Model Inputs section.

\textit{table_data_source} is either a string indicating the name of the file holding the table data or the name of an array. For more information on the \textit{table_data_source} argument, see the Table Data Source section.

\textit{table_control_string} is a two part string. The first character is an integer indicating the degrees of the spline interpolation (either 1 | 2 | 3). The second part of the control string consists of one or two characters (either C | L | E) indicating the type of extrapolation mode at the beginning and end of the data. For more information on the \textit{table_control_string} argument, see the Table Control String section.

The inputs to the \$table_model() function are described in more detail in the following sections.
Table Model Inputs

The $table_inputs$ are numerical expressions that are used as the independent model variables for the $table_model()$ function. They may be any valid expressions that can be assigned to an analog signal.

Table Data Source

The $table_data_source$ argument specifies the source of sample points for the $table_model()$ function. The sample points may come from two sources: files and arrays. The file source indicates that the sample points be stored in a file, while the array source indicates that the data points are stored in a set of array variables. The user may choose the data source by either providing the file name of a file source or a set of array variables as an argument to the function.

The table is created when the $table_model()$ system function is called for the first time. Any changes to the $table_data_source$ argument(s) of the $table_model()$ after the first call are quietly ignored (that is, the table model is not recreated). For a file source, each sample point of the table is represented as a sequence of numbers in the order of $X_1 \ X_2 \ .. \ X_N \ Y_i$, where $X_i$ is the coordinate of the sample point in $k^{th}$ dimension and $Y_i$ is the model value at this sample point. Each sample point must be separated by a new line. The numbers in the sequence must be separated by one or more spaces or tabs. Comments may be inserted before or after any sample point; comments must begin with `#' and end with a new line.

The data file must be in text format only. The numbers must be real or integer. The sample points can be stored in the file in any order.

Example

The following example shows the contents of a table model files with two dimensions.

```verbatim
# datafile.tbl
# 2-D table model sample example of the function
# f(x,y) = sqrt(x^2 + y^2)
#
# x y f(x,y)
-2 -2 2.828
-2 -1 2.236
-1 -1 1.414
0 0 0
0 1 1.0
```
If the source of the data is an array, a set of one-dimensional arrays that contain the data points must be passed to the $table_model()$ function. The size of these arrays is determined by the number of sample points in the table, $M$. The data are stored in the arrays such that for the $k^{th}$ dimension of the $i^{th}$ sample point, $kth\_dim\_array\_identifier[i] = X_{ik}$ and such that for the $i^{th}$ sample point $output\_array\_identifier[i] = Y_i$.

Example

For the previous table model example, the same data would be provided to the function in an array as shown in the following code fragment.

```verilog
@(initial_step) begin
  x[0]=-2; y[0]=-2; f_table[0]=2.828; // 0th sample point
  x[1]=-2; y[1]=-1; f_table[1]=2.236; // 1st sample point
end
```

Table Control String

The control string provides information on how the model should interpolate and extrapolate the table data. The control string consists of sub-strings for each dimension. Each sub-string may contain one character indicating the degree of the spine interpolation and an additional one or two characters indicating the type of extrapolation method to be used.

Table Interpolation Degree

The degree character is an integer between 1 and 3 representing the degrees of splines to be used for the interpolation. If not given, a degree of 1 (linear) is assumed. Table 10-1 shows the possible settings.

<table>
<thead>
<tr>
<th>Table Interpolation Character</th>
<th>Interpolation Character Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linear spline (degree 1)</td>
</tr>
<tr>
<td>2</td>
<td>Quadratic spline (degree 2)</td>
</tr>
<tr>
<td>3</td>
<td>Cubic spline (degree 3)</td>
</tr>
</tbody>
</table>
Extrapolation Control String

The extrapolation control string is used to control the algorithm to extrapolate beyond the supplied data domain. The string may contain one or two extrapolation method characters. The extrapolation method determines the behavior of the table model when the point to be evaluated is beyond the domain of the user provided sample points. The Clamp extrapolation method, specified with the character C, uses a constant value from the last data point to extend the model. The Linear extrapolation method, specified with the character L, uses piecewise linear interpolation to estimate the requested point. The user may also disable extrapolation by setting the Error extrapolation method using the character E. In this case, an extrapolation error is reported if the $table_model() function is requested to evaluate a point beyond the interpolation region. Table 10-2 summarizes these options.

<table>
<thead>
<tr>
<th>Table Extrapolation Character</th>
<th>Extrapolation Character Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Clamp extrapolation</td>
</tr>
<tr>
<td>L</td>
<td>Linear extrapolation (default)</td>
</tr>
<tr>
<td>E</td>
<td>Error condition</td>
</tr>
</tbody>
</table>

For each dimension of the table, users may use up to two extrapolation method characters to specify the extrapolation method used for each end of the data set. When no extrapolation method character is supplied, the Linear extrapolation method will be used for both ends as default behavior.

When a single extrapolation method character is supplied, the specified extrapolation method will be used for both ends of the data set. When two extrapolation method characters are supplied, the first character specifies the extrapolation method used for the end with the lower coordinate value and the second character specifies the extrapolation method for the end with the higher coordinate value. Table 10-3 illustrates some control strings and their interpretation.

<table>
<thead>
<tr>
<th>Control String</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>“1LE,2EC”</td>
<td>1st dimension linear interpolation, linear extrapolation on left, error on extrapolation to right</td>
</tr>
<tr>
<td>“”</td>
<td>2nd dimension quadratic interpolation, error on extrapolation to left, clamp on extrapolation to right</td>
</tr>
<tr>
<td>,2</td>
<td>Linear interpolation, Linear extrapolation to left and right</td>
</tr>
<tr>
<td>“3,1”</td>
<td>1st dimension linear interpolation, 2nd dimension quadratic interpolation, linear extrapolation to left and right</td>
</tr>
<tr>
<td>“1st dimension cubic interpolation, 2nd dimension linear interpolation, linear extrapolation to left and right</td>
<td></td>
</tr>
</tbody>
</table>

Examples
In the first example, the data from the table defined earlier is contributed across the ports. The data in both dimensions is linearly extrapolated at both ends of the data.

```verilog
module table_resistor (n1, n2);
electrical n1, n2;
analog begin
    I(n1, n2) <+ $table_model (V(n1), V(n2), "datafile.tbl", "1L,1L");
end
endmodule
```

In the second example, the same information is supplied within the module using the array method.

```verilog
module user_table(n1, n2);
electrical n1, n2;
real x[0:7], y[0:7], f_table[0:7];
analog begin
    @(initial_step) begin
        x[0]=-2; y[0]=-2; f_table[0]=2.828; // 0th sample point
        x[1]=-2; y[1]=-1; f_table[1]=2.236; // 1st sample point
    end
    I(a, b) <+ $table_model (V(n1), V(n2), x, y, f_table, "1L,1L");
end
endmodule
```

### File Input/Output Operations

There are several functions to provide reading and writing to files on the operating system. `$fopen()` opens a file for output while `$fclose()` closes the file. `$fstrobe()`, `$fdisplay`, and `$fwrite()` provide functions to write to the file.
The fopen Function

The file open function, $fopen()$, returns a value whose bits indicate a corresponding channel available for writing. $fopen()$ opens the file specified as an argument and returns the next available 32-bit multichannel descriptor, which is unique for the file. If the file could not be found or opened for writing, it returns 0.

The multichannel descriptor can be considered to be a set of 32 flags, where each flag represents a single output channel. The least significant bit (bit 0) always represents standard output, also called channel 0, while the other bits represent channels which have been opened by $fopen()$.

The first call to $fopen()$ opens channel 1 and returns a value of 2 (bit 1 of the descriptor is set). The next call to $fopen()$ opens channel 2 and returns a value of 4 (bit 2 of the descriptor is set). Subsequent calls open channels 3, 4, 5, etc. and return values of 8, 16, 32, etc. A channel number corresponds to a single bit in the multichannel descriptor. Up to 32 channels may be opened.

**Syntax**

```
multi-channel_descriptor = $fopen( file_name );
```

where `multi-channel_descriptor` is an integer value and `file_name` is the name of the file to be opened.

The fclose Function

The file close function, $fclose()$, closes the specified channel in the multichannel descriptor. Further output to the closed channel is no longer allowed. The $fopen()$ function reuses channels which have been closed.

**Syntax**

```
$fclose( multi_channel_descriptor );
```

where `multi-channel_descriptor` is an integer value representing the channel(s) to be closed.
The fstrobe Function

The file strobe function, $fstrobe(), writes date to the channel(s) specified in the multichannel descriptor.

Syntax

$fstrobe ( multi_channel_descriptor [, list_of_arguments ] );

where multi_channel_descriptor is represents one or more opened files and list_of_arguments is an optional, comma separated list of quoted strings or expressions. The arguments for list_of_arguments are the same as those available for the $strobe() function argument.

Example

integer multi_ch_desc1, multi_ch_desc2, data_value;
 @(initial_step) begin
   multi_ch_desc1 = $fopen("data1.txt");
   multi_ch_desc2 = $fopen("data2.txt");
   data_value = 1;
end
$fstrobe(multi_ch_desc1 | multi_ch_desc2,
          "Write value %d to both data1.txt and data2.txt",
          data_value) ;

The fdisplay Function

The file display function, $fdisplay(), writes date to the channel(s) specified in the multichannel descriptor. It provides the same capability as the $fstrobe() function.

Syntax

$fdisplay ( multi_channel_descriptor [, list_of_arguments ] );

where multi_channel_descriptor is represents one or more opened files and list_of_arguments is an optional, comma separated list of quoted strings or expressions. The arguments for list_of_arguments are the same as those available for the $strobe argument.
The fwrite Function

The file write function, $fwrite(), writes data to the channel(s) specified in the multichannel descriptor. It provides the same capability as the $fstrobe() function but without the newline character.

Syntax

    $fwrite ( multi_channel_descriptor [, list_of_arguments ] );

where multi_channel_descriptor is a list of one or more opened files and list_of_arguments is an optional, comma separated list of quoted strings or expressions. The arguments for list_of_arguments are the same as those available for the $fstrobe() function argument.

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 );

where list_of_arguments is a comma separated list of quoted strings or expressions.

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.

Syntax

$display( list_of_arguments );

Example

$display("\n\nWarning: parameter X is %g, max allowed is %g\n\n", X, maxX);

Debug Function

The debug function, $debug(), provides the same capability as the $strobe function, except that it is displayed at each iteration of the analog solver. It is useful for displaying information for models when the analog simulation is not converging.

Syntax

$debug( list_of_arguments );

Example

$debug("\nThe value of variable X is %g ", X);

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.
16. Escape Sequences

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\n</td>
<td>newline character</td>
</tr>
<tr>
<td>\t</td>
<td>tab character</td>
</tr>
<tr>
<td>&quot;</td>
<td>\ character</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot; character</td>
</tr>
<tr>
<td>\ddd</td>
<td>character specified by 1-3 octal digits</td>
</tr>
<tr>
<td>%%%</td>
<td>% character</td>
</tr>
</tbody>
</table>

17. Format Specifications

<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%h or %H</td>
<td>hexadecimal format</td>
</tr>
<tr>
<td>%d or %D</td>
<td>decimal format</td>
</tr>
<tr>
<td>%o or %O</td>
<td>octal format</td>
</tr>
<tr>
<td>%b or %B</td>
<td>binary format</td>
</tr>
<tr>
<td>%c or %C</td>
<td>ASCII format</td>
</tr>
<tr>
<td>%m or %M</td>
<td>hierarchical name format</td>
</tr>
<tr>
<td>%s or %S</td>
<td>string format</td>
</tr>
</tbody>
</table>

18. 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>
<tr>
<td>%r or %R</td>
<td>Display 'real' in engineering notation using scale factors</td>
</tr>
</tbody>
</table>

Simulator Control Operations

Simulator control functions provide a means to interrupt simulator execution from the module.
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.

Example

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
- `default_transition

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*$abstime);
```

results in the following code

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

The directive name must be a valid identifier. It must be a sequence of alphanumeric 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 1p // 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.
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

`installationdir/veriloga/include`

where installationdir is the installation folder for Tanner Tools. Most Verilog-A files begin by including disciplines.vams and constants.vams as

```
`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,

```
`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,

```
`ifdef macro
    statements
`endif
```

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

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

**Default transition**

The `default_transition` directive specifies the default value for the rise and fall time for the transition filter function and the default value of the transition time for the Z transform family of filter functions. The directive accepts a single real number as argument. For example,

```
`default_transition 1s
`default_transition 1ms
```

Filters following the directive use its value when their transition arguments are either not supplied or evaluate to 0.0. Each successive default transition directive updates the latest default transition to be used by the filters. There are no scope restrictions on the directive. It may be placed at any point in a Verilog-A file.

**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
`ifdef
```
\`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**

\`default_function_type_analog is 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.

<table>
<thead>
<tr>
<th>A</th>
<th>and</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>absdelay</td>
</tr>
<tr>
<td>acos</td>
<td>asin</td>
</tr>
<tr>
<td>acosh</td>
<td>asinh</td>
</tr>
<tr>
<td>ac_stim</td>
<td>assign</td>
</tr>
<tr>
<td>always</td>
<td>atan</td>
</tr>
<tr>
<td>analog</td>
<td>atan2</td>
</tr>
<tr>
<td>analysis</td>
<td>atanh</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B,C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td>casez</td>
</tr>
<tr>
<td>bound_step</td>
<td>ceil</td>
</tr>
<tr>
<td>branch</td>
<td>cmos</td>
</tr>
<tr>
<td>buf</td>
<td>connectrules</td>
</tr>
<tr>
<td>bufif0</td>
<td>cos</td>
</tr>
<tr>
<td>bufif1</td>
<td>cosh</td>
</tr>
<tr>
<td>case</td>
<td>cross</td>
</tr>
<tr>
<td>casex</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ddt</td>
<td>disable</td>
</tr>
<tr>
<td>deassign</td>
<td>discipline</td>
</tr>
<tr>
<td>default</td>
<td>discontinuity</td>
</tr>
<tr>
<td>defparam</td>
<td>driver_update</td>
</tr>
<tr>
<td>delay</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>edge</td>
<td>endnature</td>
</tr>
<tr>
<td>else</td>
<td>endprimitive</td>
</tr>
<tr>
<td>end</td>
<td>endspecify</td>
</tr>
<tr>
<td>endcase</td>
<td>endtable</td>
</tr>
<tr>
<td>endconnectrules</td>
<td>endtask</td>
</tr>
<tr>
<td>enddiscipline</td>
<td>event</td>
</tr>
<tr>
<td>endfunction</td>
<td>exclude</td>
</tr>
<tr>
<td>endmodule</td>
<td>exp</td>
</tr>
</tbody>
</table>
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_zp
  large
  last_crossing
  limexp
  ln
  log
  macromodule
  max
  medium

  min
  module
  nand
  nature
  negedge
  net_resolution
  nmos
  noise_table
  nor
  not
  notif0
  notif1

O,P
  or
  output
  parameter
  pmos
  posedge
  potential

  pow
  primitive
  pull0
  pull1
  pullup
  pulldown

R,S
  rcmos
  real
  realtime
  reg
  release
  repeat

  sin
  sinh
  slew
  small
  specify
  specparam
rnmos  sqrt
rmpos  strobe
rtran  strong0
rtranif0  strong1
rtranif1  supply0
scalared  supply1

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T</strong></td>
<td></td>
</tr>
<tr>
<td>table</td>
<td>tranif1</td>
</tr>
<tr>
<td>tan</td>
<td>transition</td>
</tr>
<tr>
<td>tanh</td>
<td>tri</td>
</tr>
<tr>
<td>task</td>
<td>tri0</td>
</tr>
<tr>
<td>temperature</td>
<td>tri1</td>
</tr>
<tr>
<td>time</td>
<td>triand</td>
</tr>
<tr>
<td>timer</td>
<td>trior</td>
</tr>
<tr>
<td>tran</td>
<td>trireg</td>
</tr>
<tr>
<td>tranif0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>V,W,X,Z</strong></td>
<td></td>
</tr>
<tr>
<td>vectored</td>
<td>wor</td>
</tr>
<tr>
<td>vt</td>
<td>wreal</td>
</tr>
<tr>
<td>wait</td>
<td>xnor</td>
</tr>
<tr>
<td>wand</td>
<td>xor</td>
</tr>
<tr>
<td>weak0</td>
<td>zi_nd</td>
</tr>
<tr>
<td>weak1</td>
<td>zi_npp</td>
</tr>
<tr>
<td>while</td>
<td>zi_zd</td>
</tr>
<tr>
<td>white_noise</td>
<td>zi_zp</td>
</tr>
<tr>
<td>wire</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B: Unsupported Elements

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

1.Unsupported Elements

| Hierarchy: | Ordered parameter lists in hierarchical instantiation |
|           | Hierarchical names, except for node.potential.abstol and node.flow.abstol, which are supported |
|           | Derived natures |
|           | The defparam statement |

| Functions: | Accessing variables defined outside function’s parent module |

| Input / Output: | Enforcement of input, output, and inout |
|                | Parameter-sized ports |

| System tasks: | $realtime scaled to the `timescale directive |
|              | $monitor |
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

/*
   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

discipline logic
domain discrete;
enddiscipline

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

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

// Charge in coulombs
nature Charge
units = "coul";
access = Q;
ddt_nature = Current;
`ifdef CHARGE_ABSTOL
abstol = `CHARGE_ABSTOL;
`else
abstol = 1e-14;
`endif
endnature
// Potential in volts
nature Voltage
units = “V”;
access = V;
idt_nature = Flux;
`ifdef VOLTAGE_ABSTOL
abstol = `VOLTAGE_ABSTOL;
`else
abstol = 1e-6;
`endif
endnature
// Flux in Webers
nature Flux
units = “Wb”;
access = Phi;
ddt_nature = Voltage;
`ifdef FLUX_ABSTOL
abstol = `FLUX_ABSTOL;
`else
abstol = 1e-9;
`endif
endnature
// Conservative discipline
discipline electrical
potential Voltage;
flow Current;
dendiscipline
// Signal flow disciplines
discipline voltage
potential Voltage;
dendiscipline
discipline current
potential Current;
dendiscipline
// 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;
`endif
endnature

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

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

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

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

// Kinematic
// Position in meters
nature Position
units = "m";
access = Pos;
ddt_nature = Velocity;
`ifdef POSITION_ABSTOL
abstol = `POSITION_ABSTOL;
`else
abstol = 1e-6;
`endif
endnature
// Velocity in meters per second
nature Velocity
units = "m/s";
access = Vel;
ddt_nature = Acceleration;
idt_nature = Position;
`ifdef VELOCITY_ABSTOL
  abstol = `VELOCITY_ABSTOL;
`else
  abstol = 1e-6;
`endif
denature

// 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
denature

// 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
denature

// Force in Newtons
nature Force
units = "N";
access = F;
`ifdef FORCE_ABSTOL
  abstol = `FORCE_ABSTOL;
`else
  abstol = 1e-6;
`endif
denature
// 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
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 0.7853981633974484830962
`define M_1_PI 0.31830988618379067154
`define M_2_PI 0.63661977236758134308
`define M_SQRT2 1.41421356237309504880
`define M_SQRT1_2 0.70710678118654752440
// P_ indicates a physical constant
// charge of electron in coulombs
`define P_Q 1.6021918e-19

// speed of light in vacuum in meters/sec
`define P_C 2.997924582e8

// Boltzman’s constant in joules/kelvin
`define P_K 1.3806226e-23

// Plank’s constant in joules*sec
`define P_H 6.6260755e-34

// permittivity of vacuum in farads/meter
`define P_EPS0 8.85418792394420013968e-12

// permeability of vacuum in henrys/meter
`define P_U0 (4.0e-7 * `M_PI)

// zero celsius in kelvin
`define P_CELSIUS0 273.15
`endif
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;
  parameter string type = "NMOS" from {"NMOS","PMOS"};
  real X;
  // this is a single line comment
  /* this is a
   * comment block */
  analog begin
    @(initial_step) begin
      // performed at the first timestep of an analysis
    end
    if (input2 > 0) begin : local_block_name
      $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
  end
endmodule
```
1. 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 [(integer</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.

2. 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>Derivative</td>
<td>The ddx operator returns the symbolic derivative of an expression relative to a state function.</td>
</tr>
<tr>
<td></td>
<td>ddt(expr, potential_or_flow(name))</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>
| Discrete waveform filters | The transition filter smooths out piecewise linear waveforms.  
\[
\text{transition}( \ expr \ [ , \ td \ [ , \ rise\_time \ [ , \ fall\_time \ [ , \ time\_tol ] ] ] ] )
\]  
The slew analog operator bounds the rate of change (slope) of the waveform.  
\[
\text{slew}( \ expr \ [ , \ max\_pos\_slew\_rate \ [ , \ max\_neg\_slew\_rate ] ] )
\]  
The last_crossing() function returns a real value representing the simulation time when a signal expression last crossed zero.  
\[
\text{last\_crossing}(expr, \ direction)
\] |
| Laplace transform filters | 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.  
\[
\text{laplace\_zp}(expr, \ z, \ r)
\] |
| 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 t0. T specifies the period of the filter, is mandatory, and must be positive. t specifies the transition time, is optional, and must be nonnegative. 
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.  
\[
\text{zi\_zp}(expr, \ z, \ r, \ T \ [ , \ t \ [ , \ t0 ] ] )
\] |
| limexp | Limits exponential argument change from one iteration to the next.  
\[
\text{limexp}(arg)
\] |
# Mathematical Functions

3. 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

4. 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 ( \pi/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>atanh(x)</td>
<td>arch-hyperbolic tangent</td>
<td>-1 &lt;= x &lt;= 1</td>
</tr>
</tbody>
</table>

AC Analysis Stimuli

5. 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>\texttt{ac_stim( [ analysis_name [ , mag [ , phase ]]])}</td>
</tr>
</tbody>
</table>

Noise Functions

6. Noise Functions
### Verilog-A Reference Manual

#### Function

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

#### Analog Events

7. 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>$\text{initial_step\ [\ (list_of_analyses)\ ]}$</td>
</tr>
<tr>
<td>Final Step</td>
<td>Event trigger at final step.</td>
</tr>
<tr>
<td></td>
<td>$\text{final_step\ [\ (list_of_analyses)\ ]}$</td>
</tr>
<tr>
<td>Cross</td>
<td>Zero crossing threshold detection.</td>
</tr>
<tr>
<td></td>
<td>$\text{cross}(expr\ [\ , \ dir\ [\ , \ time_tol\ [\ , \ expr_tol\ ]])\ ]$</td>
</tr>
<tr>
<td>Timer</td>
<td>Generate analog event at specific time.</td>
</tr>
<tr>
<td></td>
<td>$\text{timer\ (start_time\ [\ , \ period\ [\ , \ time_tol\ ]])}$</td>
</tr>
<tr>
<td>Above</td>
<td>Generate analog event at specific threshold crossing.</td>
</tr>
<tr>
<td></td>
<td>$\text{above\ (expr\ [\ , \ time_tol\ [\ , \ expr_tol\ ]])}$</td>
</tr>
</tbody>
</table>
## Timestep Control

8. Simulator Action Control Functions

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

9. 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. $debug displays at each iteration.</td>
</tr>
<tr>
<td>$display</td>
<td>$strobe(args);</td>
</tr>
<tr>
<td>$write</td>
<td>$debug displays at each iteration. $strobe(args);</td>
</tr>
<tr>
<td>$debug</td>
<td>$debug displays at each iteration. $strobe(args);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fopen</td>
<td>Open a file for writing and assign it to an associated channel. $multi-channel_desc = $fopen(&quot;file&quot;);</td>
</tr>
<tr>
<td>$fclose</td>
<td>Close a file from a previously opened channel(s). $fclose(multi-channel_desc);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$fstrobe</td>
<td>Write simulation data to an opened channel(s) when the simulator has converged. Follows format for $strobe. $fstrobe(multi-channel_desc, &quot;info to be written&quot;);</td>
</tr>
<tr>
<td>$fdisplay</td>
<td>$fstrobe(multi-channel_desc, &quot;info to be written&quot;);</td>
</tr>
<tr>
<td>$fwrite</td>
<td>$fstrobe(multi-channel_desc, &quot;info to be written&quot;);</td>
</tr>
</tbody>
</table>

Simulator Environment Functions

The environment parameter functions return simulator environment information.

10. 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. $temperature</td>
</tr>
<tr>
<td>$abstime</td>
<td>Return absolute time in seconds. $abstime</td>
</tr>
</tbody>
</table>
$vt$

$vt$ 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.

$$\text{\$vt \ [ (Temperature) ]}$$

$analysis$

Returns true (1) if current analysis matches any one of the passed arguments.

$$\text{\$analysis(str \ [ , str ])}$$

$simparam$

Returned the queried value based on string, else return default expr.

$$\text{\$simparam(str \ [ , expr ])}$$

## Module Hierarchy

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

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

### Example

```verilog
my_src #(fstart(100), ramp(z));
```
Index

$ (dollar sign) ................................................. 4
% (modulo) ....................................................... 22
%m (hierarchical format specifier) .............. 56
) (right parenthesis) ......................................... 10
@ (at-sign operator) ...................................... 41
_ (underscore)................................................ 3, 4
__VAMS_ENABLE__ (predefined macro) ............... 62
\ (accent grave).............................................. 4, 59
{ (left bracket) ................................................ 9
} (right bracket) .............................................. 9, 10
absolute tolerance ..................................... 28, 29, 31
AC analyses .................................................. 30, 31
access functions ............................................ 11, 17
  access function value .................................... 9
ambient temperature .................................... 46, 47
analog behavior .............................................. 10, 11, 20, 41
analogs ....................................................... 10, 12, 11, 27
defining the analog block .............................. 8, 9
analog events .................................................. 43, 44
analog expressions ....................................... 19
analog functions ............................................ 16, 17
analog modeling ............................................. 5
analog operators ............................................ 27, 32, 44, 45
analog procedural block flow ......................... 15
analog signals ............................................... 6, 11, 7, 43
analyses ....................................................... 6, 11, 7, 43
analysis statements ....................................... 14
attributes ..................................................... 7
automatic conversion of numeric types ............ 20
backslash ....................................................... 60
backward compatibility .................................... 4
behavioral description .................................... 8
behavioral model .......................................... 41
block item declaration .................................... 17
block naming ................................................ 11
block statements .......................................... 2
bounds ........................................................... 47
branch contribution operator ....................... 12
branch contribution statement .................... 12
branches ...................................................... 10
case sensitivity ............................................. 4
case statements ............................................ 12, 27
Chi-Square distribution ................................ 25
comma-separated lists ................................... 9
comments ...................................................... 19
`define (compiler directive) `define ......................... 59
  `else (compiler directive) `else........................ 4
  `ifdef (compiler directive) `ifdef ...................... 59
  `ifndef (compiler directive) `ifndef ............... 59
  `include (compiler directive) `include .............. 59
  `resetall (compiler directive) `resetall .......... 59
  `undef (compiler directive) `undef .................. 59
  `undef (compiler directive) `undef ............... 59
  `include (compiler directive) `include ............ 60
  `ifdef (compiler directive) `ifdef .................. 60
  `ifdef (compiler directive) `ifdef ............... 61
  `ifdef (compiler directive) `ifdef ............... 61
  `ifdef (compiler directive) `ifdef ............... 61
  `ifdef (compiler directive) `ifdef ............... 61
  `ifdef (compiler directive) `ifdef ............... 61
  `endif (compiler directive) `endif ............... 61
  `endif (compiler directive) `endif ............... 61
complex constants ....................................... 2
compound elements ...................................... 19
conditional compilation ................................ 59, 61
conditional execution .................................... 17
conditional expressions .............................. 27
conditional macros ....................................... 61
conditional statements ................................ 11, 27
constant assignment value............................ 10
constant expressions .................................... 6, 9, 12
constant numbers ........................................ 9
numbers ........................................................ 2
constants ..................................................... 2
continuous domain ........................................ 11
correlation operator ...................................... 11
correlation statements ................................... 11, 17, 45
core data ..................................................... 9
controlled transitions .................................... 30
covergence .................................................... 34
problems ..................................................... 48
properties .................................................... 27
simulator ..................................................... 55
data types .................................................... 6, 20
DC analyses ............................................... 29, 30, 31, 33, 41
operators
  absdelay ..................................................30

event or operator
  event or ..................................................45

optimization .............................................5

optional block identifier ................................11

ordered lists ................................................14, 15

default parameter values
  default parameter values ................................14

parameters
  defining parameters ..................................8, 9
  parameter description ...................................5

parameters
  parameter declaration ....................................14

parameters
  defining parameters ....................................17

input parameters
  input parameters .........................................17

parameters
  parameterized macros .....................................60

passed variable arguments ................................17

phase ..........................................................50

phase shift ..................................................30

dependence
  constant waveforms .....................................30

continuous signal ..........................................32

piecewise
  linear .....................................................31, 51

Poisson distribution .......................................24

ports
  defining ports ..........................................8, 9
  port assignment ...........................................15

positive transitions .......................................31

potential .....................................................10, 7

power ..........................................................50

power spectral density ...................................51

predefined mathematical constants
  M_1_PI ..................................................73
  M_2_PI ..................................................73
  M_2_SQRTPI .............................................73
  M_E ..........................................................73
  M_LN10 ....................................................73
  M_LN2 ....................................................73
  M_LOG10E ..................................................73
  M_LOG2E ..................................................73
  M_PI .......................................................73
  M_PI_2 ...................................................73
  M_PI_4 ...................................................73
  M_SQRT1_2 ................................................73
  M_SQRT2 ...................................................73
  M_TWO_PI ..................................................73

predefined non-escaped identifiers ......................4

predefined physical constants
  P_C .......................................................74
  P_CELSIUS0 ...............................................47, 74
  P_EPS0 ....................................................74
  P_H .......................................................74
  P_K .......................................................74
  P_Q .......................................................74
  P_U0 .......................................................74

preprocessing directives ..................................59

procedural control statements .............................11

procedural statements ......................................10

pseudo-random bit stream generator .....................45

pseudo-random numbers ....................................22

radians ......................................................21, 50

random number generation ..................................22

range ..........................................................10

real
  constants ...............................................2
  data types ...............................................6
  declarations ..............................................6

numbers
  real .......................................................3

reference net ...............................................12

relative filenames ........................................60

repeat statements ..........................................13, 27

`celldefine (reserved directive)
  `celldefine .............................................62

`default_discipline (reserved directive)
  `default_discipline .....................................62

`default_nettype (reserved directive)
  `default_nettype .........................................62

`elsif (reserved directive)
  `elsif .....................................................62

`endcelldefine (reserved directive)
  `endcelldefine ...........................................62

`ifndef (reserved directive)
  `ifndef ...................................................62

`ifndef (reserved directive)
  `ifndef ...................................................62

`line (reserved directive)
  `line .......................................................62

`nounconnected_drive (reserved directive)
  `nounconnected_drive ....................................62

`timescale (reserved directive)
  `timescale ................................................62

`unconnected_drive (reserved directive)
  `unconnected_drive .......................................62

scalar .......................................................8

elements ....................................................19

scale factors ...............................................2

scientific notation .........................................3

scope ..........................................................16

sequential block ..........................................11

short-circuiting ..........................................19

signal