SIMetrix

SIMetrix Verilog-A Manual

Version 8

August 2015

SIMetrix Verilog-A Manual

Copyright © SIMetrix Technologies Ltd. 1992-2015

SIMetrix Technologies Ltd.,
78 Chapel Street,
Thatcham,
Berkshire
RG18 4QN
United Kingdom

Tel:	+44 1635 866395
Fax:	+44 1635 868322
Email:	support@simetrix.co.uk
Web:	http://www.simetrix.co.uk

Contents

1 Introduction **1**
1.1 What Is Verilog-A? . . . 1
1.2 Verilog-A Language Reference Manual . . . 1

2 Using Verilog-A Compiler **2**
2.1 Using Verilog-A with SIMetrix Schematics . . . 2
2.2 Defining Verilog-A Files in Netlist . . . 2
2.3 Messages . . . 3
2.4 .LOAD Full Syntax . . . 3
2.5 Verilog-A Cache . . . 4
2.6 Permananent .SXDEV Installation . . . 4

3 Writing Verilog-A Code **5**
3.1 Overview . . . 5
3.2 Verilog-A Text Editor . . . 5
3.3 Hello World! . . . 5
3.4 A Simple Device Model . . . 7
3.4.1 Module Ports . . . 8
3.4.2 Branch Contributions . . . 8
3.4.3 Parameters . . . 8
3.4.4 Disciplines . . . 9
3.5 A Resistor . . . 9
3.6 A Soft Limiter . . . 10
3.6.1 Variables . . . 11
3.6.2 $finish . . . 11
3.6.3 Functions . . . 11
3.6.4 Local Parameters . . . 11
3.6.5 Parameter Limits . . . 12
3.6.6 Conditional Statements . . . 12
3.7 A Capacitor . . . 12
3.8 A Voltage Controlled Oscillator . . . 13
3.9 Digital Elements - Overview . . . 14
3.10 Digital Gate . . . 14
3.10.1 cross() Monitored Event . . . 15
3.10.2 transition() Analog Operator . . . 16
3.11 Butterworth Filter . . . 17
3.11.1 Arrays . . . 18
3.11.2 For Loops . . . 18
3.11.3 laplace_nd Function . . . 18
3.12 RC Ladder . . . 19
3.12.1 Vectors of Nodes . . . 20
3.12.2 Analog For Loops and genvars . . . 20
3.12.3 Compile-time Parameters . . . 21
3.12.4 See Also . . . 21

4 Verilog-A Reference **22**
4.1 Verilog-A Functions . . . 22
4.1.1 $abstime . . . 24
4.1.2 $bound_step . . . 24
4.1.3 $debug . . . 24
4.1.4 $discontinuity . . . 25
4.1.5 $display . . . 25
4.1.6 $fclose . . . 27
4.1.7 $fdebug . . . 27
4.1.8 $fdisplay . . . 28
4.1.9 $finish . . . 28
4.1.10 $fmonitor . . . 28
4.1.11 $fopen . . . 28
4.1.12 $fstrobe . . . 29
4.1.13 $fwrite . . . 29
4.1.14 $mfactor . . . 29
4.1.15 $monitor . . . 29
4.1.16 $param_given . . . 30
4.1.17 $port_connected . . . 30
4.1.18 $random . . . 30
4.1.19 $simparam . . . 31
4.1.20 $stop . . . 31
4.1.21 $strobe . . . 32
4.1.22 $table_model . . . 32
4.1.23 $temperature . . . 32
4.1.24 $vt . . . 32
4.1.25 $write . . . 33
4.1.26 abs . . . 33
4.1.27 absdelay . . . 33
4.1.28 ac_stim . . . 34
4.1.29 acos . . . 34
4.1.30 acosh . . . 34
4.1.31 analysis . . . 34
4.1.32 asin . . . 35
4.1.33 asinh . . . 35
4.1.34 atan . . . 35
4.1.35 atan2 . . . 35
4.1.36 atanh . . . 35
4.1.37 ceil . . . 35
4.1.38 cos . . . 35
4.1.39 cosh . . . 35
4.1.40 cross . . . 35
4.1.41 ddt . . . 36
4.1.42 ddx . . . 37
4.1.43 exp . . . 37
4.1.44 flicker_noise . . . 37
4.1.45 floor . . . 38
4.1.46 hypot . . . 38
4.1.47 idt . . . 38
4.1.48 laplace_nd . . . 38
4.1.49 laplace_np . . . 39
4.1.50 laplace_zd . . . 39
4.1.51 laplace_zp . . . 40
4.1.52 last_crossing . . . 40
4.1.53 limexp . . . 41
4.1.54 ln . . . 41

4.1.55 log . . . 41
4.1.56 max . . . 41
4.1.57 min . . . 41
4.1.58 pow . . . 41
4.1.59 sin . . . 41
4.1.60 sinh . . . 41
4.1.61 slew . . . 42
4.1.62 sqrt . . . 42
4.1.63 tan . . . 42
4.1.64 tanh . . . 42
4.1.65 timer . . . 42
4.1.66 transition . . . 43
4.1.67 white_noise . . . 44
4.2 Analog Operator Restrictions . . . 44

5 Implementation - vs LRM **46**
5.1 Overview . . . 46
5.2 SIMetrix Verilog-A vs LRM 2.2 . . . 46
5.2.1 2.6 Strings . . . 46
5.2.2 2.8.1 . . . 46
5.2.3 3.2.2 Parameters - Value Range Specification . . . 46
5.2.4 3.2.3 Parameter Units and Descriptions . . . 46
5.2.5 3.4.2.2 Domain Binding . . . 47
5.2.6 3.4.2.3 Empty Disciplines . . . 47
5.2.7 3.4.2.4 Disciplines of Wires and Undeclared Nets . . . 47
5.2.8 3.4.2.7 User Defined Attributes . . . 47
5.2.9 3.4.3.1 Net Descriptions . . . 47
5.2.10 3.4.3.2 Net Discipline Initial (Nodeset) Values . . . 47
5.2.11 3.4.5 Implicit Nets . . . 47
5.2.12 3.5 Real Net Declarations . . . 47
5.2.13 3.6 Default Discipline . . . 47
5.2.14 3.7 Discipline Precedence . . . 47
5.2.15 3.8 Net compatibility . . . 47
5.2.16 3.9 Branches . . . 48
5.2.17 4.1.6 Case Equality Operator . . . 48
5.2.18 4.1.13 Concatenations . . . 48
5.2.19 4.2.3 Error Handling . . . 48
5.2.20 4.4.1 Restrictions on Analog Operators . . . 48
5.2.21 4.4.4 Time derivative Operator . . . 48
5.2.22 4.4.5 Time integral operator . . . 48
5.2.23 4.4.6 Circular Intergral Operators . . . 49
5.2.24 4.4.13 Z-transform filters . . . 49
5.2.25 4.5.1 Analysis . . . 49
5.2.26 4.5.2 DC analysis . . . 49
5.2.27 4.5.4.3 Noise_table . . . 49
5.2.28 4.6.1 Defining an Analog Function . . . 49
5.2.29 4.6.2 Returning a Value from an Analog Function . . . 49
5.2.30 5.3.2 Indirect Branch Assignments . . . 49
5.2.31 6.7.5. Above Function . . . 49
5.2.32 7 Hierarchical Structures . . . 49
5.2.33 8 . . . 50
5.2.34 9 Scheduling Semantics . . . 50
5.2.35 10.1 Environment Parameter Functions . . . 50
5.2.36 10.2 $random Function . . . 50
5.2.37 10.2 $dist_ Functions . . . 50
5.2.38 10.4 Simulation Control System Tasks . . . 50

5.2.39 10.7 Announcing Discontinuity . . . 50
5.2.40 10.9 Limiting Functions . . . 50
5.2.41 10.10 Hierarchical System Parameter Functions . . . 50
5.2.42 11.1 ‘default_discipline . . . 50
5.2.43 11.2 ‘default_transition . . . 50
5.2.44 11.6 ‘resetall . . . 51
5.2.45 11.7 Pre-defined Macros . . . 51
5.2.46 12, 13 . . . 51
5.3 SIMetrix Extensions . . . 51
5.3.1 In an Ideal World... . . . 51
5.3.2 Analog Operator Syntax . . . 51
5.3.3 Instance Parameters . . . 52
5.3.4 Device Mapping . . . 52
5.3.5 Tolerances . . . 53
5.3.6 Analysis() Function . . . 53
5.3.7 $simparam() Function . . . 53
5.3.8 $fopen() Function . . . 53
5.4 Verilog-A Interaction . . . 54
5.4.1 Real-Time Noise . . . 54
5.4.2 Transient Snapshots . . . 54
5.4.3 Pseudo-Transient Analysis . . . 54

Chapter 1

Introduction

1.1 What Is Verilog-A?

Verilog-A is a language for defining analog models; it is suitable for defining behavioural models with a high level of abstraction as well as highly detailed models for semiconductor devices.

Prior to the introduction of Verilog-A and other similar languages (e.g. VHDL-AMS and MAST), the definition of such models could only be achieved, if at all, using subcircuits of controlled sources, arbitrary sources and various semiconductor devices. This method is inflexible, clumsy and usually very inefficient.

Further, SIMetrix Verilog-A is a compiled language. This means that the Verilog-A code is compiled to a binary executable program in the same way that built-in device models are implemented. This makes Verilog-A models very fast.

The SIMetrix implementation of Verilog-A uses a compiler to translate the Verilog-A source into program code using the 'C' language. This in turn is compiled into a DLL which is then loaded into the SIMetrix memory image. Access to the verilog-A description is then made at the netlist level using models and instance lines.

You do not need to install a 'C' compiler to use Verilog-A. SIMetrix Verilog-A is supplied with the open-source 'C' compiler gcc using the mingw extensions. We have used a stripped down version of gcc that includes only the essential files needed for this.

The SIMetrix Verilog-A compiler was developed by us; we do not license a third-party's product, nor is it based on open-source software. This means that we know it inside out and will be able to offer the same high level of support that we have always offered with all our products.

1.2 Verilog-A Language Reference Manual

The language reference manual may be obtained from *Verilog-A Language Reference Manual*

The SIMetrix implementation is based on version 2.2.

Chapter 2

Using Verilog-A Compiler

2.1 Using Verilog-A with SIMetrix Schematics

SIMetrix has a simple feature that will create a schematic symbol for use with a Verilog-A definition. The feature invokes the Verilog-A compiler using an option that tells it just to execute the first part of the compilation process. This allows the script to learn some information about the Verilog-A file such as module and port names. The script will ask you where you wish each pin to be located and after that will create a symbol and place it on the schematic. The symbol will be decorated with all necessary properties to interface the Verilog-A model to the simulator.

To use the script, create a Verilog-A definition, then execute the schematic menu **Verilog-A | Construct Verilog-A Symbol**. Navigate to the Verilog-A file (extension .va) then close. Select pin locations as requested. Image of symbol will appear for placement.

The symbol can be found for future use using **Place | From Symbol Library** then navigate to "Auto Created Symbols -> Verilog-A Symbols".

2.2 Defining Verilog-A Files in Netlist

Use the simulator statement '.LOAD' to specify the Verilog-A source file. E.g.:

```
.LOAD resistor.va
```

This will invoke the Verilog-A compiler (va.exe) which will create one common 'C' file and one 'C' file per module statement within the Verilog-A file. The 'C' files will then be compiled and linked using gcc to produce the final DLL which has the extension .sxdev. These files are all placed in the directory %APPDATAPATH%\SIMetrix600\vacache where %APPDATAPATH% is your application data directory.

Having compiled the va file, .LOAD will load the .sxdev file into the SIMetrix memory image. It will then map the code within into the simulator's model table making the new device ready for use.

To use the new device or devices, defined with Verilog-A module statements, you must specify a .MODEL statement. These must be placed *after* the .LOAD statement. The format of the .MODEL statement should be:

```
.MODEL modelname va-mod-name parameters
```

Where *modelname* is the model name referred to on the instance (see below), *va-mod-name* is the name of the module in the Verilog-A source file and *parameters* are parameters defined using the Verilog-A parameter keyword.

To create instances of the new device create an instance line (or schematic symbol with appropriate properties) that begins with one of the letters 'N', 'P', 'W', 'U' or 'Y'. You can use other letters as long as the number of terminals is compatible with the original use of that letter. For example, you can use the letter 'M' as long as the device has four terminals - as a MOS device would have. But you must use one of 'N', 'P', 'W', 'U' or 'Y' for devices with more than 4 terminals or only a single terminal. 'Q' will work for three or four terminals, but you should avoid using it as it is full of amiguities as a result of SPICE history.

When you start a new simulation, any sxdev files loaded in the previous run will be unloaded and the model table entries removed.

2.3 Messages

When running a simulation, you will see a number of messages in the command shell. These are output by the VA compiler, the MAKE utility and, if you are unlucky, the 'C' compiler.

Errors or warning output by the Verilog-A compiler will be displayed during this process.These will be in the form:

```
*** ERROR *** (@'verilog-a-filename',linenum), error-message
```

If the problem is with the syntax, the message will say *** SYNTAX ERROR ***.

NOTE: Identifiers that you use in your Verilog-A code (e.g. variables, parameters, ports etc) may be prefixed with an underscore when referenced in any warning or error message.

When you run a .VA file for the sescond and subsequent time without editing it, you will not see any messages from the Verilog-A compiler.

2.4 .LOAD Full Syntax

```
.LOAD file [instparams=parameter_list] [nicenames=0|1] [goiters=goiters]
[ctparams=ctparams] [suffix=suffix] [warn=warnlevel]
```

file can specify either a Verilog-A fileor a .SXDEV file. If the extension is .SXDEV, no compilation will be performed and the specified file will be loaded directly. The remaining options described above will not be recognised in this case. Otherwise the build sequence described above will be initiated. Paths are relative to the current working directory. **Don't use .VA file names containing spaces.**

parameter_list is a list of parameter name separated by commas. There should be no spaces in this list. Each parameter in this list will be defined as an instance parameter. See "Instance Parameters" on page 52 for details.

goiters specifies the number of global optimiser iterations. The default is 3. A higher number may improve the execution speed of the code at the expense of a longer compilation time. In practice this will only have a noticeable effect on very large verilog-a files. Setting the value to zero will disable the global optimiser. This is likely to slow execution speed a little. The global optimiser is an algorithm that cleans up redundant statements in the 'C' file.

ctparams defines 'Compile-time parameters' and is a list of comma-separated parameter name/value pairs in the form *name=value*. Any parameters listed will be substituted with the constant value defined during compilation as if it were entered as a literal constant in the verilog-a code. This feature is especially useful for items such as array sizes and vectored port sizes. A considerably more efficient result will be produced if the values of such items are known at compile time.

warnlevel sets a filter for warning messages. If set to zero, no warnings will be displayed. If set to 2, all warnings will be displayed. The default is 1 which will cause most warnings to be displayed but will omit those that are less serious.

nicenames=0|1 is an advanced feature for debugging purposes. It tells the compiler to use meaningful names in the 'C' file if possible. Otherwise it will used short names. There is a small risk of a name clash in the 'C' file if this option is switched on.

2.5 Verilog-A Cache

SIMetrix will reuse existing Verilog-A binary files without recompiling if the source files have not changed. It determines whether or not the file has changed by calculating an MD5 checksum on the source files and comparing this with a value stored in the .sxdev file. While this method is slower than the more conventional method of checking file dates, it is more robust and reliable.

This cache mechanism can save significant time if the VA definition is large. The hicum model, for example, takes about 6 seconds to compile.

You can clear the cache at any time using the schematic menu **Verilog-A | Clear Cache**. This will delete all files in the cache directory.

2.6 Permananent .SXDEV Installation

The .SXDEV files may be relocated to the plugins\devices directory in which case they become a built-in device. Currently, you should not expect binary compatibility between versions. A Verilog-A license is required to load a .sxdev file.

Chapter 3

Writing Verilog-A Code

3.1 Overview

We will introduce Verilog-A by showing a number of examples. Each example introduces a new concept or language feature. This is not a definitive reference of the language but we hope to demonstrate the most commonly used features. The table below lists the examples used in this manual along with the path of the files where you can find a read-to-run schematic and Verilog-A definition file.

Example	File Location
Hello World!	Examples/Verilog-A/Manual/Hello-world
A Simple Device Model	Examples/Verilog-A/Manual/Gain-block
A Resistor	Example/Verilog-A/Manual/Resistor
A Soft Limiter	Example/Verilog-A/Manual/Soft-limiter
A Capacitor	Example/Verilog-A/Manual/Capacitor
A Voltage Controlled Oscillator	Example/Verilog-A/Manual/Vco
Digital Gate	Example/Verilog-A/Manual/Gates
Butterworth Filter	Example/Verilog-A/Manual/Butterworth-filter
RC Ladder - Loops, Vectored Nodes and genvars	Example/Verilog-A/Manual/RC-ladder

3.2 Verilog-A Text Editor

SIMetrix includes a built-in Verilog-A text editor configured for Verilog-A syntax highlighting. To open a new Verilog-A design, select menu **File | New Verilog-A**. To open an existing file, you can drag and drop the file into the command shell message window or use menu **File | Open...**. Select VerilogA files from the drop-down box in the bottom right corner.

3.3 Hello World!

It has become customary to introduce any computer language with a "Hello world" program. That is a program that simply prints "Hello World!". While Verilog-A was not designed to perform

this type of task, it is nevertheless possible to write such a program. Here is an example:

```
module hello_world ;

    analog
    begin

        @(initial_step)
            $strobe("Hello World!") ;
    end
endmodule
```

You can try this using the following procedure:

1. Open a new Verilog-A design using menu **File | New Verilog-A** then enter the lines above. (This will copy and paste from the PDF OK, but be aware that in general copying and pasting ASCII text from PDFs can result in strange problems. In particular, watch out for '-' characters. These aren't always what they seem.)
2. Save to a file called hello_world.va
3. Start SIMetrix if you have not already done so. Open an empty schematic sheet.
4. Save the schematic to any file name.
5. Select menu **Verilog-A | Construct Verilog-A Symbol**
6. Navigate to the file you created in step 2 above
7. Select OK
8. Place symbol that is created. It's just a box with no pins
9. Setup a transient analysis with any stop time you like
10. Run simulation

The first time you run this, you will see messages relating to the compilation procedure. After that the message "Hello World!" will be displayed in the command shell.

If you get any error messages, check the code you entered. The error message should point to the line where the problem occurred. Be aware that sometimes the line number given may not be exact. The point where the parser detects that something is wrong may occur one or two lines after the actual cause of the problem. For example, if you omitted the ';' on the line containing the $strobe call, you would the error "Unexpected token 'end"' error reported for the following line or possibly even the line after that. The 'end' token would not be expected if the ';' was missing but this is on the next line.

Although our hello world program does not do much, it does introduce a number of Verilog-A concepts:

1. Modules. All devices that can be instantiated as models and instances are defined as modules. In the above example the module has the name `hello_world`. This name is used in the associated .MODEL statement in the SIMetrix netlist to access this module.
2. The *analog block* denoted by the keyword `analog`. This is where the main body of the Verilog-A definition is placed
3. Initial step *event* denoted by `@(initial_step)`. The statement following this will be executed only in the first step of the simulation, that is, the dc operating point phase. You might like to see what happens if you remove this line. You can do this easily by 'commenting it out' which can be done with to forward slashes like this:

```
//@(initial_step)
```

4. $strobe. This is known as a *system task* in the Verilog-A LRM (language reference manual). $strobe outputs a message to the command shell. It can also output values in various format and behaves in a similar way to the 'C' printf function. We will see more of this later.

3.4 A Simple Device Model

We will now show how to make a simple gain block. Here is the Verilog-A design:

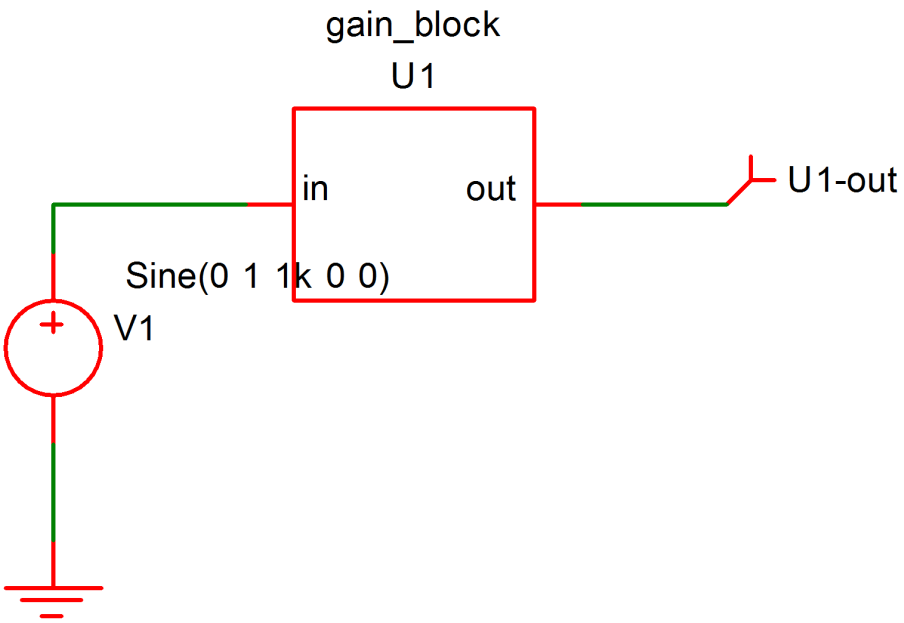
```
`include "disciplines.vams"
module gain_block(in, out) ;

    electrical in, out ;
    parameter real gain=1.0 ;

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

endmodule
```

You may like to enter the above example in the same manner as the previous hello world example. We suggest entering this circuit:



When using the menu **Verilog-A | Construct Verilog-A Symbol** for the above definition, you will notice a dialog box appear asking the location for the device's pins. You would not have seen this with the hello world example as that device does not have any pins. Choose 'left' for 'in' and 'right' for 'out'.

We have supplied the above pre-built. See Examples/Manual/gain-block. All the examples used in this manual are available from Examples/Manual. However, you may find it more instructive to enter the code and schematics manually. But we would advise against copying and pasting from this document as some characters may not copy correctly. We suggest type in by hand - note that

the quote character before `include` is a 'back tick' - usually the key at the far left of the row of numbers on US, UK and other keyboards.

Run the above in the usual way. You should see an output that follows the input; that is a 1V 1kHz sine wave.

3.4.1 Module Ports

In the above definition we have introduced two 'module ports' to the module definition. These define connection terminals and the generated symbol shows these as pins 'in' and 'out'.

3.4.2 Branch Contributions

The line:

```
V(out) <+ V(in)*gain  ;
```

defines the relationship between the module ports `out` and `in`. This is known as a *branch contribution* and is one of the most important Verilog-A concepts. Branch contributions define a relationship that the simulator must maintain between the 'probes' (`V(in)` in the above) on the right hand side and the 'source' (`V(out)` above) on the left hand side. They behave in the same way as arbitrary source devices. The above, for example, is equivalent to a SIMetrix netlist line like this:

```
B1 out 0 v = V(in) * gain
```

Branch contributions, however, differ from arbitrary sources in that they are additive. Successive branch contributions with the same left hand side add to each other. This applies to both voltage and current sources. For example:

```
V(out) <+ V(in)*gain  ;
V(out) <+ 1.0 ;
```

is the same as:

```
V(out) <+ V(in)*gain + 1.0 ;
```

The V() function in the above is known as an *access function*. Access functions may have one or two arguments each of which must refer to a port or internal node. If two arguments are provided, V() accesses the potential between the two nodes. If only a single node is supplied, it accesses the potential between that node and ground.

The access function I() access the current flowing between its two nodes. As with the voltage access function, if only a single node is provided, the second node is implicitly ground.

The access functions V() and I() are not defined as language keywords but are in fact defined by the `electrical` discipline contained within the `disciplines.vams` file.

3.4.3 Parameters

```
parameter real gain=1.0 ;
```

defines a parameter and gives it a default value of 1.0. This value can be edited at the netlist level. If you used a generated symbol or our pre-built example, double click the device U1, then enter:

```
gain=5
```

Now rerun the schematic. Notice the output amplitude increase to 5V peak.

3.4.4 Disciplines

Finally, you will notice two other lines not in the hello world example:

```
electrical in, out ;
```

defines the discipline for the module ports in this case 'electrical'. Verilog-A supports other disciplines such as thermal, mechanical and rotational allowing simulation of physical processes other than electrical and electronic. The definitions of these other disciplines are defined in the disciplines.vams file which is included using the line:

```
`include "disciplines.vams"
```

Nearly all Verilog-A definitions include this line at the top of the file. We excluded it from the hello world example as that did not need it.

3.5 A Resistor

In this example we define a simple resistor. A resistor is a device whose current is proportional to the voltage difference between its terminals. This is defined in Verilog-A using a branch contribution as follows:

```
I(p,n) <+ V(p,n)/resistance ;
```

This defines the current/voltage relationship that the simulator must maintain on the nodes `p` and `n`. `I(p,n)` represents the current flowing from port `p` to port `n` and `V(p,n)` represents the potential difference measured between nodes `p` and `n`.

Here is the full definition:

```
`include "disciplines.vams"

module va_resistor(p,n) ;

    parameter real resistance = 1000.0 from (0.0:inf] ;
    electrical p, n ;

    analog
        I(p,n) <+ V(p,n)/resistance ;

endmodule
```

In the above the `resistance` parameter has been given value range limits to prevent resistance value of zero or below. A `resistance` of zero would lead to a divide-by-zero error.

Instead of blocking resistors with a value of zero, we could instead implement a zero resistance using a zero voltage contribution. This is how:

```
`include "disciplines.vams"

module va_resistor(p,n) ;

    parameter real resistance = 1000.0 ;
    electrical p, n ;

    analog
```

```
    begin
        if (resistance!=0.0)
            I(p,n) <+ V(p,n)/resistance ;
        else
            V(p,n) <+ 0.0 ;
    end

endmodule
```

Note the conditional statement starting `if (resistance!=0.0)`. Notice also, that the analog block is now enclosed with the keywords `begin` and `end`. These are not actually necessary in this case, but are necessary where there is more than one statement in the analog block.

3.6 A Soft Limiter

This is a definition for a soft limiter device. This will pass the input signal through unchanged up to some limit after which it will follow a decaying exponential in the form:

```
1 - exp( -(v-vlim) )
```

The same in reverse occurs for the lower limit. Here is the full definition:

```
`include "disciplines.vams"
module soft_limiter(in, out) ;

    electrical in, out ;
    parameter real  vlow=-1.0,
                    vhigh=1.0,
                    soft=0.1 from (0:1.0) ;

    localparam real band = (vhigh-vlow)*soft,
                    vlow_1 = vlow+band,
                    vhigh_1 = vhigh-band ;

    real vin ;

    analog
    begin

        @(initial_step)
            if (vhigh<vlow)
            begin
                $strobe("Lower limit must be less than higher limit") ;
                $finish ;
            end

        vin = V(in) ;

        if (vin>vhigh_1)
            V(out) <+ vhigh_1+band*(1.0-exp(-(vin-vhigh_1)/band));
        else if (vin<vlow_1)
            V(out) <+ vlow_1-band*(1.0-exp((vin-vlow_1)/band)) ;
        else
            V(out) <+ vin ;
    end
endmodule
```

See Examples/Manual/Soft-limiter example

The above example introduces the following new concepts:

1. Variables. We use `vin` to hold the value of `V(in)`. In this example we have done this simply to make the code a little more readable. But variables can store any value or expression and have a much wider use
2. The `$finish` system task.
3. The `exp` function
4. Local parameters using the `localparam` keyword
5. Parameter value range limits using the `from` keyword. (Also used in the resistor previous)

The soft limit example also uses a conditional statements using `if` and `else` which we first saw with the resistor example above.

3.6.1 Variables

Variables, such as `vin` in the example must be declared first. In the above example this declaration is the line:

```
real vin ;
```

This declares the variable 'real'. This is 'real' in the computing sense meaning that the value is stored using floating point arithmetic and can take non-integer values. The alternative declaration is `integer` which means the variable stores whole numbers. Variable declarations, like parameter declarations must be placed within the `module` - `endmodule` block. They can be declared outside the analog block, as in the example above, or they can be declared inside a named `begin` - `end` block. For example

```
begin : main
    real vin ;
    ...
end
```

If declared this way, the variable may only be used within the `begin` - `end` block in which it was declared.

3.6.2 $finish

The `$finish` system task aborts the simulation unconditionally.

3.6.3 Functions

Verilog-A has a range of mathematical functions built-in. In the above example we have used the `exp` function. See "Verilog-A Functions" on page 22 for a complete list.

3.6.4 Local Parameters

A local parameter is one that cannot be changed by the user via the .MODEL statement or any other means. Local parameters are a way of defining constant vaues as, unlike variables, they cannot be assigned except in their declaration. In our example we declared the `band` local parameter as:

```
localparam real band = (vhigh-vlow)*soft
```

We could just as simply have defined a variable to do this. However, by using a local parameter we know it can't be subsequently modified. This aids readability but more importantly tells the compiler that it cannot change allowing to optimise the result effectively.

3.6.5 Parameter Limits

Parameters can be given maximum and minimum limits. This is done using the `from` keyword. In the above example:

```
soft=0.1 from (0:1.0)
```

defines the limits for `soft` from 0 to 1.0 *exclusive*. This means that any value greater than 0 and less than 1.0 will be accepted but the values 0 and 1.0 will not be allowed. You can also define *inclusive* limits using a square bracket instead of a round parenthesis. E.g in the following 1.0 is allowed:

```
soft=0.1 from (0:1.0]
```

3.6.6 Conditional Statements

Conditional statements are in the form:

```
if (conditional-expression)
    statement ;
else
    statement ;
```

statement may be a single statement such as a branch contribution or it may be a collection of statements enclosed by `begin` and `end`.

3.7 A Capacitor

To implement a capacitor we need a time derivative function. In Verilog-A this is achieved using the `ddt` analog operator. A capacitor can be defined using the branch contribution statement:

```
I(p,n) <+ capacitance * ddt( V(p,n)) ;
```

Like the resistor, this defines the current/voltage relationship that the simulator must maintain on the nodes `p` and `n`. However, this definition has time dependence.

Here is the complete definition for a capacitor:

```
`include "disciplines.vams"

module va_capacitor(p,n) ;

    parameter real capacitance = 1n  ;
    electrical p, n ;

    analog
        I(p,n) <+ capacitance * ddt(V(p,n)) ;

endmodule
```

See Examples/Manual/Capacitor. Note there is another definition for a capacitor with an initial condition parameter - capacitor_with_ic.va. This uses the time integration operator `idt` which allows the specification of an initial condition.

3.8 A Voltage Controlled Oscillator

Verilog-A may be used to create signal sources. Here we show how to make a voltage controlled oscillator.

```
`include "disciplines.vams"
`include "constants.vams"

module vco(in, out) ;

    parameter real  amplitude = 1.0,
                    centre_frequency = 1K,
                    gain = 1K ;

    parameter integer steps_per_cycle=20 ;

    localparam real omegac = 2.0 * `M_PI * centre_frequency,
                    omega_gain = 2.0 * `M_PI * gain ;

    electrical in, out ;

    analog
    begin : main

        real vin, instantaneousFreq ;

        vin = V(in) ;
        V(out) <+ amplitude*sin(idt(vin*omega_gain+omegac,0.0)) ;

        // Use $bound_step system task to limit time step
        // This is to ensure that sine wave is rendered with
        // adequate detail.
        instantaneousFreq = centre_frequency + gain * vin ;
        $bound_step (1.0 / instantaneousFreq / steps_per_cycle) ;
    end
endmodule
```

This can be found in Examples/Manual/Vco

This model uses the `idt` analog operator to integrate frequency to obtain phase. The frequency is calculated from `omegac` which is the constant term and `vin*omega_gain` which the voltage controlled term.

A problem with sinusoidal signals is that in order to obtain adequate resolution, the time step must be limited to a controlled fraction of the cycle time. In the above the parameter steps_per_cycle is used to define a minimum number of steps per cycle. This is implemented using the `$bound_step` system task. This tells the simulator the maximum time step it can use for the next time point. It can use a smaller step if needed but must not use a larger step.

The above can suffer a problem if left to run for a very large number of cycles. The return value from the `idt` operator increases continuously and eventually the size of this value will impact on the calculation precision available leading to inaccuracy. The problem can be resolved using the `idtmod` operator. However, this language feature has not yet been implemented in the SIMetrix Verilog-A. We expect to offer this in a later release.

3.9 Digital Elements - Overview

Verilog-A can model digital devices as well as analog. This is useful in situations where some simple logic function interfaces mostly with analog elements. An example is a phase detector in a phase-locked loop. At least one of its inputs would often come from an analog source and its output would usually drive a low-pass filter also implemented with analog components. Some phase detector designs employ a digital state machine that would usually suit a digital event driven simulator. But if it interfaces with analog devices, interface bridges would need to be connected to the analog signals. This complicates and slows down the simulation. Using Verilog-A we can efficiently implement the entire function in the analog domain.

We have provided an example of a phase detector; see Examples/phase_detector.

In this section we will show how to create some simple logic elements.

3.10 Digital Gate

Here is a definition for an AND gate

```
`include "disciplines.vams"

module and_gate(in1, in2, out);

    electrical in1, in2, out ;

    parameter real  digThresh = 2.0,
                    digOutLow = 0.0,
                    digOutHigh = 5.0,
                    trise=10n,
                    tfall=10n ;

    analog
    begin : main

        integer dig1, dig2, logicState ;

        // Detect in1 threshold
        @ (cross(V(in1)-digThresh, 0, 1n))
            if (V(in1)>digThresh)
                dig1 = 1 ;
            else
                dig1 = 0 ;

        // Detect in2 threshold
        @ (cross(V(in2)-digThresh, 0, 1n))
            if (V(in2)>digThresh)
                dig2 = 1 ;
            else
                dig2 = 0 ;

        logicState = dig1 && dig2 ? digOutHigh : digOutLow ;
        V(out) <+ transition(logicState , 0.0, trise, tfall) ;
    end
endmodule
```

This example introduces two new concepts:

1. The `cross` event
2. The `transition` analog operator

3.10.1 cross() Monitored Event

The `cross` event function is used to detect when an input signal crosses its logic threshold. Consider the line:

```
@ (cross(V(in1)-digThresh, 0, 1n))
```

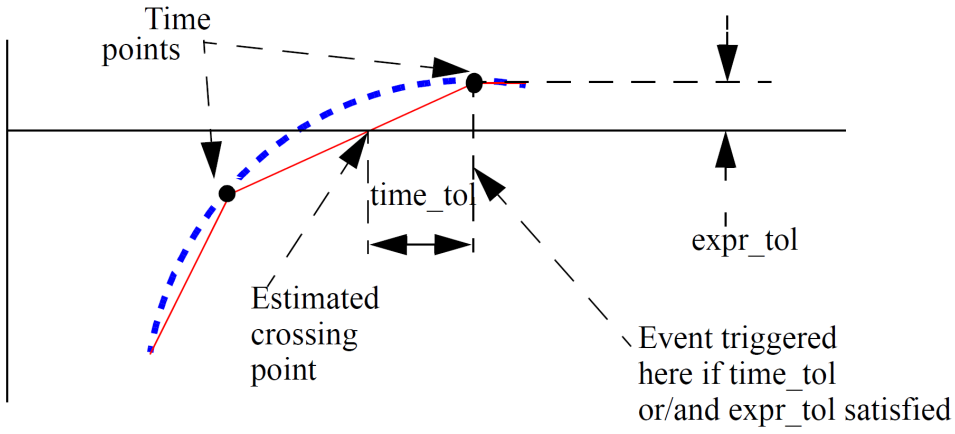
This line both defines the event and also responds to the event when it is triggered. The arguments define the event, while the statement that follows it is the action taken when the event is triggered.

The function has the following form:

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

Only the first argument is compulsory.

expr	expression to test. The event is triggered when the expression crosses zero.
edge	0, +1 or -1 to indicate edge. +1 means the event will only occur when expr is rising, -1 means it will only occur while falling and 0 means it will occur on either edge. Default=0 if omitted.
time_tol	Time tolerance for detection of zero crossing. Unless the input is moving in an exact linear fashion, it is not possible for the simulator to predict the precise location of the crossing point. But it can make an estimate and then cut or extend the time step to hit it within a defined tolerance. *time_tol* defines the time tolerance for this estimate. The event will be triggered when the difference between the current time step and the estimated crossing point is less than *time_tol*. If omitted or zero or negative, no timestep control will be applied and the event will be triggered at the first natural time point after the crossing point. See diagram below for an illustration of the meaning of this parameter.
expr_tol	Similar to *time_tol* but instead defines the tolerance on the input expression. See dialog below.



Cross Event Function Behaviour

3.10.2 transition() Analog Operator

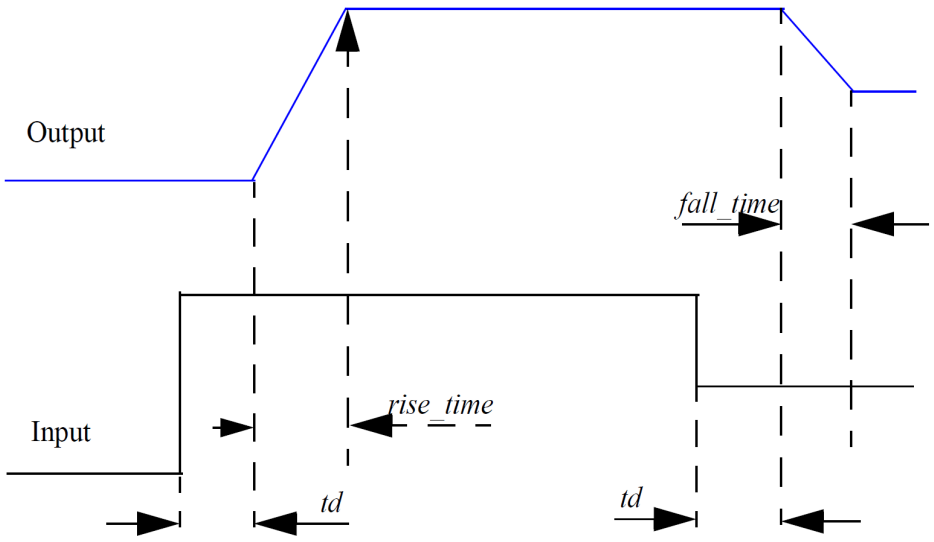
The `transition` function at the end is one of a class of functions called analog operators. The `ddt` and `idt` functions seen earlier are also analog operators. The `transition` analog operator is designed to handle signals that change in discrete steps such as the output of logic devices and digital to analog converters. In the and gate example above, the output logic level can change instantaneously but the output of a real device would typically follow a specified rise or fall time. The transition analog operator converts the discrete input value to a continuous output value using specified rise and fall times. The function has the following form:

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

expr	Input expression.
td	Delay time. This is a *transport* or *stored* delay. That is, all changes will be faithfully reproduced at the output after the specified delay time, even if the input changes more than once during the delay period. This is in contrast to *intertial* delay which swallows activity that has a shorter duration than the delay. Default=0.
rise_time	Rise time of output in response to change in input.
fall_time	Fall time of output in response to change in input.
time_tol	Ignored. The LRM does not explicitly state what this is supposed to do and we see no purpose for a tolerance parameter.

If *fall_time* is omitted and *rise_time* is specified, the *fall_time* will default to *rise_time*. If neither is specified or are set to zero, a minimum but non-zero time rise/fall time is used. This is set to the value of MINBREAK which is the minimum break point value. Refer .OPTIONS in the *Simulator Reference Manual/Command Reference/.OPTIONS* for details of MINBREAK.

The `transition` analog operator should not be used for continuously changing input values; use the `slew` or `absdelay` analog operators instead.



Transition Analog Operator Waveforms

3.11 Butterworth Filter

Here we present a butterworth filter with arbitrary order. SIMetrix already has something like this built-in, but we show a Verilog-A version to demonstrate arrays, looping constructs and the Laplace analog operators.

The design allows the user to specify the order of the filter using a model parameter. The filter itself is implemented using the analog operator `laplace_nd` which provides a Laplace transfer function defined by its numerator and denominator polynomial coefficients. To calculate the coefficients for the specified order, we build an array for the denominator coefficients using a for loop. The array only needs to be calculated once so we put this calculation in response to an initial_step event. (Actually it will be recalculated on each dc operating point iteration which is not as efficient as it could be. This is an area that we hope to address in a future revision.)

```
`include "disciplines.vams"
`include "constants.vams"

module laplace_butter(in,ref,out) ;

    real res ;
    electrical in, ref, out ;
    parameter freq=1.0 ;
    parameter integer order=5 ;

    real scale, bPrev ;
    // Denominator array size
    real den[order:0] ;
    integer k ;

    analog
    begin

        //      Calculate Butterworth coefficients
        @ (initial_step)
        begin
            scale = 1.0/freq/2/`M_PI ;
            bPrev = 1.0 ;
            den[0] = 1.0 ;

            for (k=1 ; k<order+1 ; k=k+1)
            begin
                bPrev = scale*cos((k-1.0)/order*(`M_PI*0.5))/
                        sin((k*0.5)/order*`M_PI) * bPrev ;
                den[k] = bPrev ;

                $strobe("den coeff \%d = \%g", k, den[k]) ;
            end
        end

        // Actual Butterworth filter
        res = laplace_nd( V(in,ref), {1.0}, den) ;
        V(out,ref) <+ res ;
    end
endmodule
```

See Examples/Manual/Butterworth-filter

This design introduces these language features:

1. Array variables
2. For loops
3. The `lapalace_nd` analog operator

3.11.1 Arrays

Verilog-A supports arrays of both variables and parameters. In the example above we use an array to store the denominator coefficients for the `laplace_nd` analog operator. Array variables must be declared with their range of allowed indexes using this syntax:

```
type array_name[low_index:high_index] ;
```

Where:

type	`real` or `integer`
array_name	name of array
low_index	Minimum index allowed
high_index	Maximum index allowed

low_index and *high_index* determine the number of elements in the array to be *high_index* − *low_index* + 1.

3.11.2 For Loops

For loops use a syntax similar to the 'C' language. This is as follows:

```
for (initial_assignment ; test_expression ; loop_assignment )
    statement
```

initial_assignment	Assignment statement (in the form *variable* = *expression*) that is executed just once on entry to the loop. Typically this would be an assignment that assigns a loop counter variable a constant value. In the example it assigns 1 to the variable k.
test_expression	Expression is evaluated at the start of each iteration around the loop before *statement*. If the result of the evaluation is non-zero, *statement* will be executed. If not the loop will be terminated.
loop_assignment	Assignment statement that is executed after *statement*. Typically this would be an assignment that increments or decrements a loop counter variable.In the above it increments k by 1.

3.11.3 laplace_nd Function

The laplace_nd function implements a Laplace transfer function. This is in the form:

$$H(s) = \frac{n_0 + n_1 s + n_2 s^2 + ... + n_m s^m}{d_0 + d_1 s + d_2 s^2 + ... + d_m s^m}$$

where $d_0, d_1, d_2, ..., d_m$ are the denominator coefficients and $n_0, n_1, n_2, ..., n_m$ are the numerator coefficients and the order is m.

The laplace_nd function is in the form:

`laplace_nd` (expr, num_coeffs, den_coeffs, ϵ)

Where

expr	Input expression
num_coeffs	Numerator coefficients. This can be entered as an array variable or as an array initialiser. An array initialiser is a sequence of comma separated values enclosed with '{' and '}'. E.g: { 1.0, 2.3, 3.4, 4.5}. The values do not need to be constants.
den_coeffs	Denominator coefficients in the same format as the numerator - see above. In the example this is provided as the array den. The values in den are calculated in the for loop.
ϵ	Tolerance parameter currently unused.

If the constant term on the denominator (d0 in equation above) is zero, the laplace function must exist inside a closed feedback loop. With a zero denominator, the DC gain is infinite; by putting the function inside a loop, the simulator can maintain the input at zero providing a finite output. A singular matrix error will result otherwise.

3.12 RC Ladder - Loops, Vectored Nodes and genvars

Verilog-A allows definitions to contain repeated elements defined using vectors of nodes. Here we present an example that defines an RC network with any number of elements.

```
`include "discipline.h"

/* Model for an n-stage RC ladder network */
module rc_ladder(inode[0], inode[n]) ;

    electrical [0:n] inode ;

    /* The compile_time attribute is a SIMetrix extension and is
       not part of the Verilog standard. compile_time parameters
       must be defined at the time the module is compiled. Their
       values can be specified on the .LOAD line in the netlist
       using the "ctparams" parameter. E.g. ctparams="n=8"
       If not specified on the .LOAD line, the default value
       specified here will be used. */
    (* type="compile_time" *) parameter integer n=16 ;
    parameter r=1k ;
    parameter c=1n ;

    genvar i ;

    analog
    begin
        for (i=0 ; i<=n-1 ; i=i+1)
        begin
            I(inode[i],inode[i+1]) <+(V(inode[i],inode[i+1]))/r;
            I(inode[i+1]) <+ ddt(V(inode[i+1])*c) ;
        end
    end
endmodule
```

This design introduces the following language features:

1. Vectors of nodes
2. Analog for loops and genvars
3. Compile-time parameters. (This is a SIMetrix extension and not part of the Verilog-A specification)

3.12.1 Vectors of Nodes

Verilog-A allows nodes to be specified as vectors. This can be used to implement devices that have multiple inputs or outputs (such as ADCs and DACs) as well as devices like the above example which has multiple internal elements.

The Verilog-A specification allows the size of vectored nodes to be specified as a parameter that can be assigned at run time. SIMetrix *does* allow this in some simple cases but this would not be accepted in the above example. Usually, however, vectored node sizes (`n` in the above example) are specified as a constant to be available at compile time. This can be done in a number of ways:

1. As a pre-processor constant such as

   ```
   `define n 16
   ```

   this must then be accessed using the back tick character, i.e. `‘n`.
2. As a constant `localparam` parameter. These may not be set by the user and so are fixed in value at compile time.
3. As a compile-time parameter. See below for details.

Vectors of nodes can be specified in the node discipline declaration. In the example above, this is the line:

```
electrical [0:n] inode ;
```

The nodes are accessed using square brackets enclosing a constant expression in the same way that array variables are used. For example, inode[0] is the first node in the vectored node inode while inode[n] is the last.

3.12.2 Analog For Loops and genvars

We saw for-loops in the butterworth filter example. Analog for loops are syntactically identical but use a special type of variable called a *genvar* instead of a normal variable. Analog for loops are the only type of loop where you can iterate through vectored nodes. They are also the only type of loop where you can use analog operators.

genvars are inherited from the Verilog-A version 1.0 concept called *generate statements*. Generate statements define a method of replicating a statement any number of times while increasing or decreasing a controlling variable - the *generate variable* or *genvar*. In computer science this technique is often called *loop-unrolling*. Generate statements are now considered obsolete and have been replaced by analog for loops but the functionality is similar.

The Verilog-A language specification does not stipulate that analog for loops should be unrolled but it does impose a number of restrictions on the use of genvars to make unrolling possible as long as all constant values are available at compile time. Unrolling loops that refer to vectored nodes is considerably more efficient than evaluation at run-time.

SIMetrix will unroll analog for loops if it can. If it can’t, because one or more values in the for-loop could not be evaluated at compile-time, it will still attempt to implement the design but this

process may fail in which case an error message will be displayed. If it succeeds, a level 2 warning will be raised advising that the design would be more efficient if some variables were constant.

3.12.3 Compile-time Parameters

Compile-time parameters are a SIMetrix extension and not part of the language specification. Compile-time parameters may be assigned in the .LOAD statement in the netlist or they may be defined using an attribute in the Verilog-A code, or both. This concept is in its infancy and we hope to develop it further. The attribute in the code (this is the `(* type="compile_time" *)` prefixing the parameter keyword) declares the parameter as compile-time and provides a default value. The value may be overridden in the .LOAD statement in the netlist.

3.12.4 See Also

...the DAC example at Examples/DAC. This has a vectored module port with a size that can be specified at run time via a model parameter.

Chapter 4

Verilog-A Reference

The official definition of the Verilog-A language can be found in the Language Reference Manual version 2.2 which may be obtained from here: *Verilog-A Language Reference Manual*.

Here we present descriptions of all the functions that SIMetrix currently supports.

4.1 Verilog-A Functions

Name	Return type	In types	Implemented?
$abstime	real	()	Yes
$angle	real	()	No
$bound_step	none	(real)	Yes
$debug	none	([real/integer/string...])	Yes
$discontinuity	none	([integer])	Yes
$display	none	([real/integer/string...])	Yes
$fclose	none	(integer)	Yes
$fdebug	none	(integer, [real/integer/string...])	Yes
$fdisplay	none	(integer, [real/integer/string...])	Yes
$finish	none	([integer])	Yes
$fmonitor	none	(integer, [real/integer/string...])	Yes
$fopen	integer	(string)	Yes
$fstrobe	none	(integer, [real/integer/string...])	Yes
$fwrite	none	(integer, [real/integer/string...])	Yes
$hflip	real	()	No
$limit	real	(access_func,string,real...)	No
$mfactor	real	()	Yes
$monitor	none	([real/integer/string...])	Yes
$param_given	integer	identifier	Yes
$port_connected	integer	identifier	Yes
$random	integer	(integer,[string])	Yes

Name	Return type	In types	Implemented?
$rdist_chi_square	real	(integer,real, [string])	No
$rdist_erlang	real	(integer,real,real, [string])	No
$rdist_exponential	real	(integer,real, [string])	No
$rdist_normal	real	(integer,real,real, [string])	No
$rdist_poisson	real	(integer,real, [string])	No
$rdist_t	real	(integer,real, [string])	No
$rdist_uniform	real	(integer,real,real,[string])	No
$realtime	real	([real])	No
$simparam	real	(string,[real])	Yes
$stop	none	([integer])	Yes
$strobe	none	([real/integer/string...])	Yes
$table_model	real		Yes
$temperature	real	()	Yes
$vflip	real	()	No
$vt	real	([real]	Yes
$write	none	([real/integer/string...])	Yes
$xposition	real	()	No
$yposition	real	()	No
above	integer	(real,[real,[real]])	No
abs	copies args	(real/int)	Yes
absdelay	real	(real, real, [real])	Yes
ac_stim	complex	(string,[real,[real]])	Yes
acos	real	(real)	Yes
acosh	real	(real)	Yes
analysis	integer	(string,[...])	Yes
asin	real	(real)	Yes
asinh	real	(real)	Yes
atan	real	(real)	Yes
atan2	real	(real,real)	Yes
atanh	real	(real)	Yes
ceil	real	(real)	Yes
cos	real	(real)	Yes
cosh	real	(real)	Yes
cross	integer	(real,[integer,[real,[real]]])	Yes
ddt	real	(real,[real/string])	Yes
ddx	real	(real,access_func)	Yes
exp	real	(real)	Yes
flicker_noise	real	(real,real,[string])	Yes
floor	real	(real)	Yes
hypot	real	(real,real)	Yes

Name	Return type	In types	Implemented?
idt	real	(real,[real,[real,[real/string]]])	Yes
idtmod	real	(real,[real,[real,[real,[real/string]]]])	No
laplace_nd	real	(real,real-array,real-array,[real/string])	Yes
laplace_np	real	(real,real-array,real-array,[real/string])	Yes
laplace_zd	real	(real,real-array,real-array,[real/string])	Yes
laplace_zp	real	(real,real-array,real-array,[real/string])	Yes
last_crossing	real	(real,integer)	Yes
limexp	real	(real)	Yes
ln	real	(real)	Yes
log	real	(real)	Yes
max	copies args	(real/int,real/int)	Yes
min	copies args	(real/int,real/int)	Yes
noise_table	real	(real-array,[string])	No
pow	real	(real,real)	Yes
sin	real	(real)	Yes
sinh	real	(real)	Yes
slew	real	(real,[real,[real]])	Yes
sqrt	real	(real)	Yes
tan	real	(real)	Yes
tanh	real	(real)	Yes
timer	integer	(real,[real,[real]])	Yes
transition	real	(real,[real,[real,[real,[real]]]])	Yes
white_noise	real	(real,[string])	Yes
zi_nd	real	(real,real-array,real-array,real,[real,[real]])	No
zi_np	real	(real,real-array,real-array,real,[real,[real]])	No
zi_zd	real	(real,real-array,real-array,real,[real,[real]])	No
zi_zp	real	(real,real-array,real-array,real,[real,[real]])	No

4.1.1 $abstime

```
real time = $abstime ;
```

In transient analysis, returns the absolute simulation time in seconds. In all other analyses returns zero.

4.1.2 $bound_step

```
$bound_step( expression ) ;
```

Does not return a value.

In transient analysis, instructs simulator to limit the next timestep to the value of expression.

4.1.3 $debug

```
$debug( list_of_arguments ) ;
```

Does not return a value.

$debug is a display function that displays information in the command shell. See $display for a description of its arguments. The $debug function writes to the command shell on every iteration. By contrast, other display functions such as $display only write information when an iteration has converged.

See Also

"$fdebug" on page 27

"$display" on page 28

4.1.4 $discontinuity

```
$discontinuity [ ( constant_expression ) ] ;
```

Does not return a value.

Currently $discontinuity performs no action.

4.1.5 $display

```
$display( list_of_arguments ) ;
```

Does not return a value.

$display displays text in the command shell when the current iteration converges.

The arguments can be any sequence of strings, integers or reals. The function will display these values in the order in which they appear. The values will be output literally except for the interpretation of special characters that may appear in string values. The special characters are backslash ('\') and percent ('%'). '\' is used to output special characters according to the following table:

\n	Newline character
\t	Tab character
\\	Literal \character
\"	" character
\ddd	Character specified by the ASCII code of the 1-3 octal digits

The '%' character must be followed by a character sequence that defines a format specification. In execution, the '%' and the format characters are substituted for the next value in the argument list, formatted according to the string. User's conversant with the 'C' programming language will have seen this method in the printf function. For example, %d specifies that an integer be displayed in decimal format. So, if count has a value of 453, the following:

```
$display("Count=\%d", count) ;
```

would display:

```
Count=453
```

in the command shell.

The following table shows the format codes available:

%h or %H	Hexadecimal format
%d or %D	Decimal format
%o or %O	Octal format
%b or %B	Binary format
%c or %C	ASCII character. E.g a value of 84 would display an uppercase 'T'
%m or %M	Display hierarchical name of instance. This does not use one of the subsequent arguments
%s or %S	Literal string. Expects a matching string argument
%e or %E	Real number format. See Real Number Formats below
%f or %F	Real number format. See Real Number Formats below
%g or %G	Real number format. See Real Number Formats below
%r or %R	Real number format. See Real Number Formats below

Real Number Formats

Real numbers have their own more complex format codes. These are in the form:

```
% [flag] [width] [.precision] type
```

where:

flag	Characters '-', '+', '0', space or '#'. '-' means left align the result within given width (see width) '+' means always prefix a sign even if positive '0' means prefix with leading zeros '#' forces a decimal point to be always output even if not required
width	Specifies the minimum number of characters that will be displayed, padding with spaces or zeros if needed
precision	For e and f formats (see below) specifies the number of digits after the decimal point that will be printed. If g or r format is specified, specifies the maximum number of significant digits. Default if omitted is 6.
type	e, E, f, F, g, G or r, R e or E: Signed value displayed in exponential format. E.g. 1.23456E3 f or F: Signed value in decimal format. E.g. 1234.56. Result will be very long if value is very large or very small. g or G: Uses either f or e depending on which is most compact for give precision. r or R: displays in engineering units. Uses these scale factors: T, G, M, K, k, m, u, n, p, f, a.

Notes

Currently the compiler will raise an error if the type of an argument does not match its position in a corresponding format string. For example, the following will raise an error at compile time:

```
integer count ;
...
$display("Count=%g", count) ;
```

Note that the type (i.e. integer or real) of literal constants is determined by the way they are written. If a decimal point is included or if exponential or engineering formats are used, the number is real. Otherwise it is an integer. So '11' is an integer, while 11.0 is a real.

See Also

"$fdisplay" on page 28
"$debug" on page 24
"$monitor" on page 29

4.1.6 $fclose

```
$fclose( file_descriptor ) ;
```

Does not return a value.

Closes one or more file descriptors opened with $fopen.

See Also

"$fopen" on page 28
"$fdisplay" on page 28
"$fmonitor" on page 28
"$fdebug" on page 27
"$fwrite" on page 29

4.1.7 $fdebug

```
$fdebug( file_descriptor, list_of_arguments ) ;
```

Does not return a value.

As $debug, but writes to a file or files defined by *file_descriptor*.

See Also

"$debug" on page 24
"$fopen" on page 28
"$display" on page 25
"$fdisplay" on page 28

4.1.8 $fdisplay

```
$fdisplay( file_descriptor, list_of_arguments ) ;
```

Does not return a value.

As $display, but writes to a file or files defined by *file_descriptor*.

See Also

"$display" on page 25

"$fopen" on page 28

4.1.9 $finish

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

Does not return a value.

Instructs simulator to abort. Currently the argument is ignored.

4.1.10 $fmonitor

```
$fmonitor( file_descriptor, list_of_arguments ) ;
```

Does not return a value.

As $monitor, but writes to a file or files defined by *file_descriptor*.

4.1.11 $fopen

```
integer file_descriptor = $fopen( filename ) ;
```

Returns an integer representing a multi-channel file descriptor. The descriptor can be used as an argument to $fdebug, $fdisplay, $fmonitor, $fstrobe and $fwrite to write output to a file.

There are 31 possible channels each represented by a single bit in the 32 bit returned value. The top (most significant bit) is reserved. The bottom (least siginificant) is used for standard output - i.e. displays to the command shell. Each new call to $fopen will assign the next channel and set the relevant bit. By or'ing together the results from multiple calls to $fopen, it is possible to write to more than one file at a time.

SIMetrix has a special extension to this function providing access to the list file. Use the filename "<listfile>" and the descriptor returned will access it. The following code for example will create a file descriptor that will provide writes to both the list file and a user file:

```
fd = fopen(" < listfile > "); fd = fd|fopen( "a_text_file.txt") ;
```

Further, by or'ing with 1 the file descriptor will also write to the command shell.

The file descriptor should be closed with $fclose.

See Also

"$fclose" on page 27

"$fdisplay" on page 28

4.1.12 $fstrobe

```
$fstrobe( file_descriptor, list_of_arguments ) ;
```

Does not return a value.

As $strobe, but writes to a file or files defined by *file_descriptor*. Note that the $strobe and $display functions are identical. For detailed documentation see "$display" on page 25.

See Also

"$strobe" on page 32

"$display" on page 25

"$fopen" on page 28

4.1.13 $fwrite

```
$fwrite( file_descriptor, list_of_arguments ) ;
```

Does not return a value.

As $write, but writes to a file or files defined by *file_descriptor*. Note that the $write function is identical to $display, except that it does add a new line character. For detailed documentation see "$display" on page 25.

See Also

"$write" on page 33

"$display" on page 25

"$fopen" on page 28

4.1.14 $mfactor

```
real value = $mfactor ;
```

$mfactor does not take any arguments.

Returns the scaling factor applied to the instance. The scaling factor may be set using the $mfactor parameter or using a subcircuit multiplier M. If both are used, the final scale factor will be the product of these. Refer to the LRM for more details.

The LRM currently stipulates that compilers should raise an error if $mfactor is used inappropriately. This is not currently implemented and $mfactor may be used for any purpose.

4.1.15 $monitor

```
$monitor( list_of_arguments ) ;
```

Does not return a value.

$monitor behaves in a similar manner to $display except that it only outputs a result when there is a change. In other words, the behaviour is the same as $display except that successive repeated messages will not be output.

See Also

"$fmonitor" on page 28

"$display" on page 25

4.1.16 $param_given

```
integer value = $param_given( parameter_name ) ;
```

parameter_name must be a parameter defined using the `parameter` keyword. Returns a non-zero number if *parameter_name* has been specified in a .MODEL statement or on the instance line where relevant.

4.1.17 $port_connected

```
integer value = $port_connected( port_name[ [index_expression] ] ) ;
```

Returns a non-zero value if the specified *port_name* is connected externally. If the port is vectored, then *index_expression* defining the element within the vector must also be specified. No error will be raised if the index supplied is out of range; the function will simply return false (zero).

Currently, this function will only deem a port to be unconnected if no node is specified for it in the instance netlist line. It will return true (non-zero) if a node name is supplied on the netlist line but is not connected to any other component in the netlist. For example, consider a model for a four-terminal BJTwith nodes 'C', 'B', 'E' and 'S' where 'S' is the substrate connection:

```
Q1 C B E S bjtmodelname
```

In the above the substrate connection is the node S. In this case $port_connected(S) would return true regardless of whether or not S was connected to anything else. Now consider the three terminal case:

```
Q1 C B E bjtmodelname
```

In this case the substrate connection has been omitted from the netlist line and $port_connected will return false (zero).

4.1.18 $random

```
integer value = $random [ (seed) ] ;
```

Returns a random number. This has three modes of operation according what if anything is supplied for *seed*.

Mode 1: no seed

$random will return a new random number on each call with the system choosing the seed when random is used for the first time.

Example:

```
value = $random ;
```

Mode 2: constant seed

seed may be either a literal constant or a constant expression dependent only on literal constants and parameters. In this mode $random will return a fixed random value which will not update.

Example:

```
parameter seed=23 ;
...
value = $random(seed) ;
```

Mode 3: initialised integer variable seed

In this mode the *seed* variable will be updated on each call and a new random number will be generated. The sequence of random numbers will thus be repeatable given the same initial value for *seed*.

Example:

```
real seed ;
...
@(initial_step)
    seed = 23 ;
...
value = $random(seed) ;
```

In the above, the value of `seed` will be updated each time random is called.

4.1.19 $simparam

```
real value = $simparam( string [ , default_value ] ) ;
```

Returns the value of a simulator parameter defined by *string*. Possible values of *string* are described below. If an unknown string is supplied, $simparam will return the value of *default_value* if given. If no *default_value* value is given, a run-time error will be raised.

The $simparam strings supported by SIMetrix are given as:

"gdev"	Conductance added in junction GMIN stepping algorithm
"gmin"	Value of GMIN options parameter
"simulatorSubversion"	Minor version of SIMetrix simulator. E.g. for version 6.00, result will be 0, for 6.10 result will be 10 etc
"simulatorVersion"	Major version of SIMetrix simulator. For version 6.00 this will be 6
"sourceScaleFactor"	Scale factor used for sources in source stepping algorithm and pseudo transient analysis algorithm
"tnom"	Value of TNOM options parameter
"ptaScaleFactor"	Scale factor used for pseudo transient analysis algorithm
"option_name"	Any name that may be used in a .OPTIONS statement and which has a real value

The first six items in the above follow recommended names in the Verilog-A LRM. The remainder are special to SIMetrix.

4.1.20 $stop

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

The function does not return a value. Pauses simulation after completion of current step and leaves the simulator in the same state as if the user pressed the pause button.

The argument n currently has no effect.

See Also

"$finish" on page 28

4.1.21 $strobe

```
$strobe( list_of_arguments ) ;
```

Does not return a value.

Identical to the $display function.

See Also

"$fstrobe" on page 29

"$display" on page 25

4.1.22 $table_model

```
real value =
    $table_model( table_inputs, table_data_source, table_control_string) ;
```

Please refer to section 10.12 of the Verilog-AMS Language Reference Manual version 2.2. This can be obtained from http://www.eda.org/verilog-ams/htmlpages/lit.html. SIMetrix implements the LRM specification in full.

$table_model is subject to the same restrictions as analog operators according to the Verilog-A version 2.2 LRM (See "Analog Operator Restrictions" on page 44). This restriction has been removed from the version 2.3 standard and indeed there is no fundamental reason for the restriction as the `$table_model` function does not need to store state information. Currently, SIMetrix complies with LRM 2.2 - i.e the restrictions apply - but we plan to change this with a future release.

4.1.23 $temperature

```
real value = $temperature ;
```

Returns the current simulation temperature in Kelvin.

4.1.24 $vt

```
real value = $vt [ ( temperature_expression ) ] ;
```

Returns the thermal voltage at *temperature_expression*. If *temperature_expression* is not supplied, the value at the current simulation temperature will be returned.

The thermal voltage is defined as

$$\frac{KT}{q}$$

Where, K is boltzmann's constant, T is temperature (defined by *temperature_expression*) and q is the charge on an electron. The values used for K and q are those that are used for other simulator models and are the best values known at the time the original SPICE program was developed. Since that time the accepted values for K and q have been altered slightly.

The values used are:

$K = 1.3806226\text{e-}23$

$q = 1.6021918\text{e-}19$

Currently accepted values:

$K = 1.3806504\text{e-}23$

$q = 1.60217646\text{e-}19$

4.1.25 $write

```
$write( list_of_arguments ) ;
```

Does not return a value.

$write is identical to the $display function except that it does not add a new line character at the end of the text. A new line may be explicitly inserted using the '\n' sequence.

See Also

"$fwrite" on page 29

"$display" on page 25

4.1.26 abs

```
real value = abs( x ) ;
```

Returns the absolute value of x.

4.1.27 absdelay

```
real value = absdelay( expression, tdelay [ , maxdelay ] ) ;
```

Applies a transport delay to an input signal.

expression	Input signal to delay.
tdelay	Delay in seconds. If *maxdelay* is not supplied, only the value of *tdelay* at the start of the simulation will be used and subsequent changes will be ignored. Otherwise changes to *tdelay* will be used as long as they do not exceed *maxdelay*.
maxdelay	Maximum delay permitted. If omitted changes to *tdelay* will be ignored. See *tdelay* above.

In DC analyses, *tdelay* is ignored and the return value of absdelay is *expression*. In AC analysis, the signal defined by *expression* is phase-shifted according to:

$$\text{output}(\omega) = \text{input}(\omega)\exp(-j\omega \cdot \text{tdelay})$$

In transient analysis, the signal is delayed by an amount equal to the instantaneous value of *tdelay* as long as it is positive and less than *maxdelay*. absdelay stores the past history of *expression* up to *maxdelay* so that it can retrieve the requested delayed point instantaneously.

absdelay is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"slew" on page 42

"transition" on page 43

4.1.28 ac_stim

```
real value = ac_stim( [analysis_name_string [ , mag [ , phase ]]] ) ;
```

Provides a stimulus for AC analysis, essentially identical the AC spec for a standard SPICE voltage source or current source.

analysis_name_string	Analysis name in which source is to be active. Currently this must be set to "ac" or be omitted altogether.
mag	*Magnitude* of source
phase	*Phase* of source in radians

4.1.29 acos

```
real value = acos( x ) ;
```

Returns inverse cosine in radians of x. Input range is +/- 1.

4.1.30 acosh

```
real value = acosh( x ) ;
```

Returns the inverse hyperbolic cosine of x. Range is 1.0 to ∞.

4.1.31 analysis

```
integer value = analysis( analysis_list ) ;
```

Returns non-zero if the current analysis matches any of the analysis names in the argument list. *analysis_list* is a list of strings as defined in the following table.

"static"	Any analysis that solves a DC operating point. This includes the operating point analyses carried before other analyses such as transient. It also includes DC sweep
"tran"	Transient analysis. Includes the transient analysis used for pseudo transient analysis
"ac"	AC analysis
"dc"	DC sweep
"noise"	Noise analysis not including real time noise
"tf"	Transfer fumction analysis
"pz"	Pole zero analysis
"sens"	Sensitivity analysis
"ic"	The dc operating point analysis that precedes a transient analysis
"smallsig"	Any small signal analysis: AC, noise and TF
"rtn"	Real-time noise analysis
"pta"	Pseudo transient analysis

4.1.32 asin

```
real value = asin (x ) ;
```

Returns the inverse sine in radians of x. Range is +/- 1.0.

4.1.33 asinh

```
real value = asinh( x ) ;
```

Returns the inverse hyperbolic sine of x. Range is $-\infty$ to $+\infty$.

4.1.34 atan

```
real value = atan( x ) ;
```

Returns the inverse tangent in radians of x. Range is $-\infty$ to $+\infty$.

4.1.35 atan2

```
real value = atan2( x, y ) ;
```

Returns the inverse tangent in radians of x/y. The function will return a meaningful value when y is zero.

4.1.36 atanh

```
real value = atanh( x ) ;
```

Returns the inverse hyperbolic tangent of x. Range is +/- 1.0.

4.1.37 ceil

```
real value = ceil( x ) ;
```

Returns the next integer value greater than x.

See Also

4.1.38 cos

```
real value = cos( x ) ;
```

Returns the cosine of x expressed in radians. Range is $-\infty$ to $+\infty$.

4.1.39 cosh

```
real value = cosh( x ) ;
```

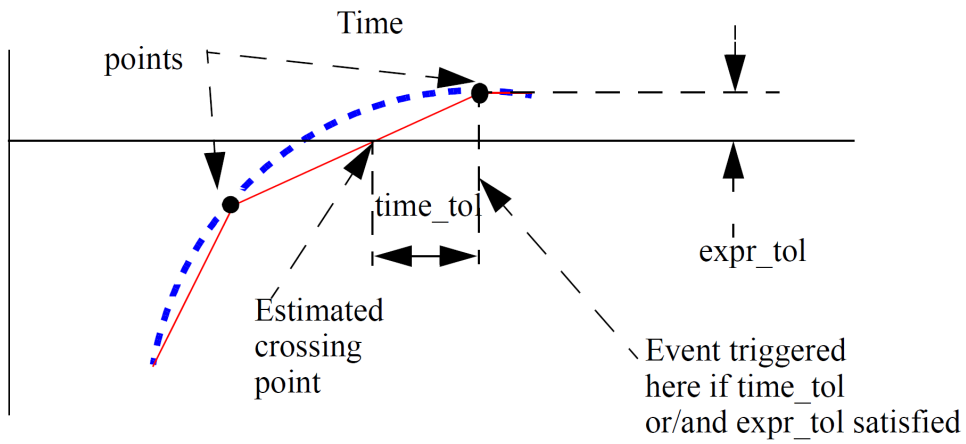
Returns the hyperbolic cosine of x. Range is approx -709 to +709.

4.1.40 cross

```
cross( expression [,edge [,time_tol [,expr_tol ]]])
```

cross is an event function and may only be used in event expressions.

expression	expression to test. The event is triggered when the expression crosses zero.
edge	0, +1 or -1 to indicate edge. +1 means the event will only occur when expr is rising, -1 means it will only occur while falling and 0 means it will occur on either edge. Default=0 if omitted.
time_tol	Time tolerance for detection of zero crossing. Unless the input is moving in an exact linear fashion, it is not possible for the simulator to predict the precise location of the crossing point. But it can make an estimate and then cut or extend the time step to hit it within a defined tolerance. *time_tol* defines the time tolerance for this estimate. The event will be triggered when the difference between the current time step and the estimated crossing point is less than *time_tol*. If omitted or zero or negative, no timestep control will be applied and the event will be triggered at the first natural time point after the crossing point. See diagram below for an illustration of the meaning of this parameter.
expr_tol	Similar to *time_tol* but instead defines the tolerance on the input expression. See figure below.



Cross Event Function Behaviour

`cross` stores state information in the same way as an analog operator. It is therefore subject to "Analog Operator Restrictions" on page 44.

See Also

"timer" on page 42

"transition" on page 43

4.1.41 ddt

```
real value = ddt( expression ) ;
```

Returns the time derivative of *expression*:

$$\frac{d}{dt}\text{expression}$$

In DC analyses, ddt returns zero. In AC analysis, the function is defined by the relation:

$$\text{output}(\omega) = input(\omega) \cdot j\omega$$

ddt is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"idt" on page 38

4.1.42 ddx

```
real value = ddx( expression, unknown_variable) ;
```

Performs symbolic differentiation on *expression* with respect to *unknown_variable*, which must be defined in terms of an access function in one of the following forms:

```
potential_access_identifier( net_or_port_scalar_expression )
```

OR

```
flow_access_identifer( branch_identifier )
```

A *potential_access_identifier* is defined in the discipline declarations and is usually 'V' for the electrical discipline. Similarly, the flow_access_identifier is usually 'I' for the electrical discipline.

net_or_port_scalar_expression can be a module port node or an internal node.

branch_identifier can be a branch defined with the `branch` keyword or an unnamed branch specifying the nodes connected to the branch.

4.1.43 exp

```
real value = exp( x ) ;
```

Returns the exponential of x. Range is $-\infty$ to about 709.

See Also

"limexp" on page 41

4.1.44 flicker_noise

```
real value = flicker_noise( power, exp [, name]) ;
```

flicker_noise is only active in small-signal noise analysis and real-time noise analysis; in other analysis modes it returns zero. It creates a noisy signal with a power of *power* at 1Hz which varies in proportion to $1/f^{\text{exp}}$.

name may be used to combine noise sources in the output report and vectors. Noise sources with the same *name* in the same instance will be combined together.

In real-time noise analysis flicker_noise simply returns a random number whose statistical distribution satisfies the characteistic of $1/f$ noise. In small signal analysis flicker_noise defines a $1/f$ noise source that may be propagated to any output node.

See Also

"white_noise" on page 44

4.1.45 floor

```
real value = floor( x ) ;
```

returns the next lower integer to x.

See Also

"ceil" on page 35

4.1.46 hypot

```
real value = hypot( x, y ) ;
```

Returns $\sqrt{x^2 + y^2}$.

4.1.47 idt

```
real value = idt( expression [, initial_condition ] ) ;
```

Returns the time integral of *expression*.

initial_condition if supplied, sets the value of the function for DC analyses including the dc operating point that precedes other analyses.

If *initial_condition* is not supplied, idt must exist inside a closed feedback loop. With no initial condition the DC gain of idt is infinite; by putting the function inside a loop, the simulator can maintain the input at zero providing a finite output. A singular matrix error will result otherwise.

idt is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"ddt" on page 36

4.1.48 laplace_nd

```
real value = laplace_nd(expr, num_coeffs, den_coeffs [, ϵ]) ;
```

Where

expr	Input expression.
num_coeffs	Numerator coefficients. This can be entered as an array variable or as an array initialiser. An array initialiser is a sequence of comma separated values enclosed with '' and ''. E.g: 1.0, 2.3, 3.4, 4.5. The values do not need to be constants.
den_coeffs	Denominator coefficients in the same format as the numerator - see above.
ϵ	Tolerance parameter currently unused.

The laplace_nd analog operator implements a Laplace transfer function. This is in the form:

$$H(s) = \frac{n_0 + n_1 \cdot s + n_s \cdot s^2 + ... + n_m \cdot s^m}{d_0 + d_1 \cdot s + d_2 \cdot s^2 + ... + d_m \cdot s^m}$$

where $d_0, d_1, d_2, ..., dm$ are the denominator coefficients and $n_0, n_1, n_2, ..., n_m$ are the numerator coefficients and the order is m.

If the constant term on the denominator (d_0 in equation above) is zero, the laplace function must exist inside a closed feedback loop. With a zero denominator, the DC gain is infinite; by putting the function inside a loop, the simulator can maintain the input at zero providing a finite output. A singular matrix error will result otherwise.

laplace_nd is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"laplace_np" on page 39

"laplace_zd" on page 39

"laplace_zp" on page 40

4.1.49 laplace_np

```
real value = laplace_np(expr, num_coeffs, poles [, ϵ]) ;
```

expr	Input expression
zeros	Zeros. See laplace_zp for details.
den_coeffs	Denominator coefficients. See laplace_nd for details.
ϵ	Tolerance parameter currently unused

laplace_np is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"laplace_nd" on page 38

"laplace_zd" on page 39

"laplace_zp" on page 40

4.1.50 laplace_zd

```
real value = laplace_zd(expr, zeros, den_coeffs [, ϵ]) ;
```

expr	Input expression.
zeros	Zeros. See laplace_zp for details.
den_coeffs	Denominator coefficients. See laplace_nd for details.
ϵ	Tolerance parameter currently unused.

laplace_zd is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"laplace_nd" on page 38

"laplace_np" on page 39

"laplace_zp" on page 40

4.1.51 laplace_zp

```
real value = laplace_zp(expr, zeros, poles [, ϵ]) ;
```

expr Input expression.

zeros Array of pairs of real numbers representing the zeros of the Laplace transform. Each pair consists of a real part and an imaginary part with the real part first. Each zero introduces a product term on the numerator in the form:

$$1 - \frac{s}{re + j \cdot im}$$

where *re* is the real part and *im* imaginary part. If a zero is complex (i.e. the imaginary part is non-zero) then its complex conjugate must also be present. If both real and imaginary parts are zero then the zero becomes just *s*.
The values can be entered as an array variable or as an array initialiser. An array initialiser is a sequence of comma separated values enclosed with '{' and '}'. E.g: 1.0, 2.3, 3.4, 4.5. The values do not need to be constants.

poles Array of pairs of real numbers representing the poles of the Laplace transform. Each pair consists of a real part and an imaginary part with the real part first. Each pole introduces a product term on the denominatr in the form:

$$1 - \frac{s}{re + j \cdot im}$$

where *re* is the real part and *im* imaginary part. If a pole is complex (i.e. the imaginary part is non-zero) then its conjugate must also be present. If both real and imaginary parts are zero then the pole becomes just *s*.
The values can be entered as an array variable or as an array initialiser. An array initialiser is a sequence of comma separated values enclosed with '{' and '}'. E.g: 1.0, 2.3, 3.4, 4.5. The values do not need to be constants.

ϵ Tolerance parameter currently unused.

laplace_zp is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"laplace_nd" on page 38

"laplace_np" on page 39

"laplace_zd" on page 39

4.1.52 last_crossing

```
real value = last_crossing( expression, direction ) ;
```

last_crossing returns the time in seconds when *expression* last crossed zero. First order interpolation is used to estimate the time of the crossing. *direction* controls the direction of the crossing. If +1 then the most recent positive transition is returned. If -1, the most recent negative transition and if zero the most recent in either direction is returned.

last_crossing returns a negative number if *expression* has not crossed zero since the start of the simulation. SIMetrix Verilog-A last_crossing implementation also returns a negative number for DC analyses but this is not defined in the standard.

last_crossing is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

4.1.53 limexp

```
real value = limexp( x ) ;
```

Returns the exponential of x. limexp limits its change in output from iteration to iteration in order to improve convergence. In situations where its return value is not the true exponential of x it will force further iterations. The iteration will only be accepted when the result is the true value of exp(x). Thus, limexp can be seen as a direct replacement for exp but with improved convergence. But note that limexp is an analog operator and is therefore subject to "Analog Operator Restrictions" on page 44.

See Also

"exp" on page 37

4.1.54 ln

```
real value = ln( x ) ;
```

Returns the natural logarithm of x. Range is 0.0 to ∞.

4.1.55 log

```
real value = log( x ) ;
```

Returns the logarithm to base 10 of x. Range is 0.0 to ∞.

4.1.56 max

```
real value = max( x, y ) ;
```

Returns x or y whichever is larger. Equivalent to ($x>y$? x : y)

4.1.57 min

```
real value = min( x, y ) ;
```

Returns x or y whichever is smaller. Equivalent to ($x<y$? x : y)

4.1.58 pow

```
real value = pow( x, y ) ;
```

Returns x^y. if x is less than zero, y must be an integer. If $x = 0$, y must be greater than zero.

4.1.59 sin

```
real value = sin( x ) ;
```

Returns the sine of x given in radians. Range $-\infty$ to ∞.

4.1.60 sinh

```
real value = sinh( x ) ;
```

Returns the hyperbolic sine of x. Range is approx -709 to +709.

4.1.61 slew

```
real value = slew( expression [, slew_pos [, slew_neg]] ) ;
```

Implements a slew rate limiter. *slew_pos* is expected to be positive and *slew_neg* is expected to be negative. If *slew_neg* is not specified or greater than or equal to zero, it assumes a value of -*slew_pos*. If neither *slew_pos* or *slew_neg* is present, expression is passed through to *value* unchanged.

`slew` limits the positive and negative rate of change of its return value to *slew_pos* and *slew_neg* respectively.

`slew` is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"absdelay" on page 33

"transition" on page 43

4.1.62 sqrt

```
real value = sqrt( x ) ;
```

Returns the square root of x. Range is 0 to ∞. Although valid, $x = 0$ should be avoided and if possible code included to prevent $x = 0$. This is because the first derivative of sqrt at zero is infinite and convergence at this value can be problematic.

4.1.63 tan

```
real value = tan(  x ) ;
```

Returns the tangent of x given in radians. Range is $-\infty$ to ∞.

4.1.64 tanh

```
real value = tanh( x ) ;
```

Returns the hyperbolic tangent of x. Range is $-\infty$ to ∞.

4.1.65 timer

```
timer( time [, period [, time_tol]])
```

timer is an event function and may only be used in an event statement in the form:

```
@(timer(...))
    statement ;
```

statement is executed when the event is triggered.

timer sets a future event to occur at a specified time either just once or repeating at a specified period.

The event is first scheduled at *time*. If *period* is specified and is greater then zero, subsequent events will also be scheduled at *time* + n**period* where n is a positive integer.

Usually the specified event will be scheduled at exactly the time specified. However, the analog simulator will not allow time points to be forced too close together as this can lead to numerical

problems as well as unnecessarily long simulation times. For this reason, the simulator may schedule the event slightly later than specified if the time point is to close to an existing time point, perhaps set by another device. The *time_tol* argument controls the tolerance of the event time. The simulator will always schedule the event so that it is within *time_tol* of the requested time. If *time_tol* is not specified the event will be scheduled after the requested time but not more than the amount specified by the MINBREAK simulaion parameter.

See Also

"cross" on page 35

4.1.66 transition

```
real value = transition(expr[, td[, rise_time[, fall_time [, time_tol]]]]);
```

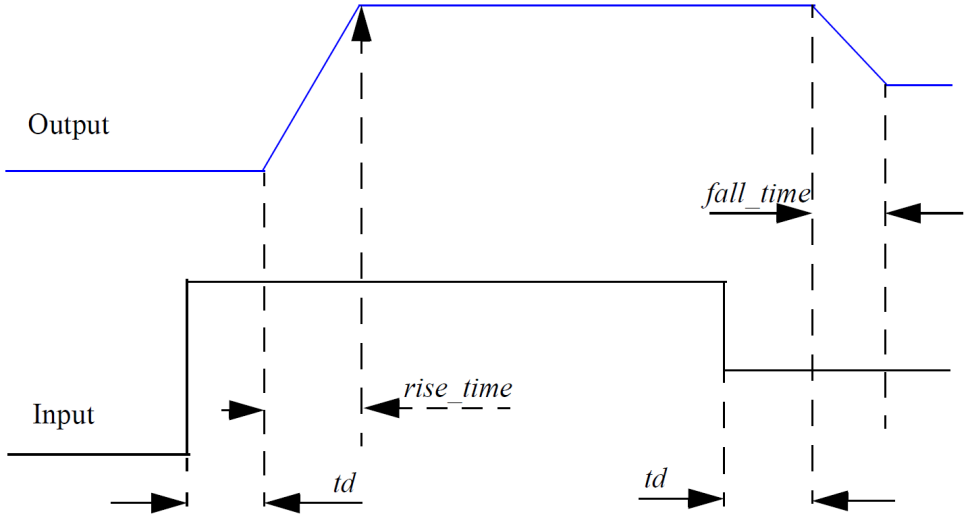
The transition analog operator converts the discrete input value to a continuous output value using specified rise and fall times.

Its arguments are:

expr	Input expression.
td	Delay time. This is a transport or stored delay. That is, all changes will be faithfully reproduced at the output after the specified delay time, even if the input changes more than once during the delay period. This is in contrast to inertial delay which swallows activity that has a shorter duration than the delay. Default=0.
rise_time	Rise time of output in response to change in input.
fall_time	Fall time of output in response to change in input.
time_tol	Currently ignored.

If *fall_time* is omitted and *rise_time* is specified, the *fall_time* will default to *rise_time*. If neither is specified or are set to zero, a minimum but non-zero time rise/fall time is used. This is set to the value of MINBREAK which is the minimum break point value. Refer .OPTIONS in the *Simulator Reference Manual/Command Reference/.OPTIONS* for details of MINBREAK.

The transition analog operator should not be used for continuously changing input values; use the slew or absdelay analog operators instead.



`transition` is an analog operator and is subject to "Analog Operator Restrictions" on page 44.

See Also

"absdelay" on page 33

"slew" on page 42

4.1.67 white_noise

```
real value = white_noise( power [, name]) ;
```

white_noise is only active in small-signal noise analysis and real-time noise analysis; in other analysis modes it returns zero. It creates a noisy signal with a power of *power* and a flat frequency distribution.

name may be used to combine noise sources in the output report and vectors for small-signal noise analysis. *name* is ignored with real-time noise analysis. Noise sources with the same *name* in the same instance will be combined together.

In real-time noise analysis white_noise simply returns a random number whose statistical distribution satisfies the characteistic of Gaussian noise. In small signal analysis white_noise defines a noise source that may be propagated to any output node.

See Also

"flicker_noise" on page 37

4.2 Analog Operator Restrictions

A number of functions are classed as *analog operators*. These functions store state information. That is, their return value depends on previous history and not just on the current value of its arguments. Because of this, analog operators are subject to some restrictions on where they may be used. These restrictions are as follows:

1. Analog operators may not be used inside a conditional statement (`if` or `case`) if the conditional expression controlling that statement could change during the course of a transient analysis. For example, the following *is not* permitted

```
if (V(n1)>0)
    I(out) <+ ddt(cap*V(out)) ;
```

   In the above `V(n1)>0` could change in a transient analysis if the voltage on node n1 rises above or below zero. This means that `ddt` would only get executed some of the time and so its state history would not always be correct.

   The following *is* permitted

```
paramter integer enable_cap = 0 ;
...
if (enable_cap)
    I(out) <+ ddt(cap*V(out)) ;
```

   this is OK because `enable_cap` is a parameter and will have a fixed value during the course of a transient analysis. So either `ddt` will always be executed or it will never be executed. Both scenarios will work correctly.

2. Analog operators are not permitted in `repeat` or `while` loops nor are they permitted in `for` loops that are not analog-for loops.

Analog operators are permitted in *analog for loops*. These are for loops controlled by a `genvar` controlling variable. This is explained in "Analog For Loops and genvars" on page 20.

The analog operator restrictions apply to the following functions:

"$table_model" on page 32 (But see note in documentation for function)

"absdelay" on page 33

"cross" on page 35

"ddt" on page 36

"idt" on page 38

"laplace_nd" on page 38

"laplace_np" on page 39

"laplace_zd" on page 39

"laplace_zp" on page 40

"last_crossing" on page 40

"limexp" on page 41

"slew" on page 42

"transition" on page 43

Chapter 5

Implementation - vs LRM

5.1 Overview

Here we describe how SIMetrix Verilog-A compares with the stadard as defined in Language Reference Manual 2.2. Full details are below: SIMetrix Verilog-A vs LRM 2.2. For SIMetrix extensions, see "SIMetrix Extensions" on page 51.

5.2 SIMetrix Verilog-A vs LRM 2.2

In the following we have highlighted areas where the SIMetrix Verilog-A compiler is not compliant with the LRM 2.2 standard.

5.2.1 2.6 Strings

String variables are not supported. This is compliant with the Annex C 'Analog Language Subset'

5.2.2 2.8.1

desc and units attributes may be included but will not be functional.

5.2.3 3.2.2 Parameters - Value Range Specification

Only the first **from** specification will be functional. Subsequent exclude specifications will be accepted by the compiler but will have no effect.

5.2.4 3.2.3 Parameter Units and Descriptions

Syntax for "desc" and "units" is recognised but non-functional.

Non-standard SIMetrix attribute "instance" has been implemented. This defines the parameter as an instance parameter, that is, its value can be set on the device line. See "Instance Parameters" on page 52.

5.2.5 3.4.2.2 Domain Binding

Anything other than “domain continuous” will raise an error.

5.2.6 3.4.2.3 Empty Disciplines

Implemented but non-functional

5.2.7 3.4.2.4 Disciplines of Wires and Undeclared Nets

Not supported

5.2.8 3.4.2.7 User Defined Attributes

Accepted but non-functional

5.2.9 3.4.3.1 Net Descriptions

Not implemented. This will lead to a syntax error if used.

5.2.10 3.4.3.2 Net Discipline Initial (Nodeset) Values

Not implemented. This will lead to a syntax error if used.

5.2.11 3.4.5 Implicit Nets

Not meaningful as hierarchical structures are not yet implemented.

5.2.12 3.5 Real Net Declarations

Not supported in Verilog-A

5.2.13 3.6 Default Discipline

Not supported in Verilog-A

5.2.14 3.7 Discipline Precedence

Not meaningful as hierarchical structures are not yet implemented.

5.2.15 3.8 Net compatibility

As hierarchical structures are not yet implemented, this is mostly not relevant.

But this is partially implemented within the simulator. If you connect different disciplines together you will get a warning. But the inherited disciplines will not be compatible, only the same disciplines may be inter-connected. ... and you only get a warning not an error.

5.2.16 3.9 Branches

Compliant for scalars only. Currently named vector branches are not supported. Unnamed branches are however fully supported.

Discipline compatibility is checked, but it seems that the discipline for each node in a branch must be identical. The spec requires them to be 'compatible' which is not the same thing.

Minor issue: if a branch is unused then the discipline of each node will not be checked at all and no error will be raised if they are incompatible. This is not defined in the standard.

5.2.17 4.1.6 Case Equality Operator

Not supported in Verilog-A

5.2.18 4.1.13 Concatenations

Array initialisers are supported. Replication multiplier is not supported.

5.2.19 4.2.3 Error Handling

Not correctly implemented.

Its possible that this may never be implemented to the letter of the standard. While attempting to iterate to convergence, it is not at all uncommon for maths functions to be overflow or to receive invalid arguments. When this happens, SIMetrix reduces the step (whatever that step may be) and tries again. This algorithm is often successful.

Complying with the most literal interpretation of this would be undesirable as it would mean some simulations failing when they may have been perfectly solveable.

5.2.20 4.4.1 Restrictions on Analog Operators

SIMetrix Verilog-A is mostly compliant with this section with the exception detailed below.

Analog operators (such as `ddt`, `transition` etc) are not allowed in places where their execution could be dependent on values that change during the course of a simulation. This is because analog operators store state information which could become invalid. SIMetrix does not always implement this restriction correctly and there are situation where it will allow you to use an analog operator but shouldn't.

5.2.21 4.4.4 Time derivative Operator

Compliant except tolerance is currently ignored.

5.2.22 4.4.5 Time integral operator

idt(expr) - compliant

idt(expr,ic) - compliant

Others not implemented.

5.2.23 4.4.6 Circular Intergral Operators

Not implemented.

5.2.24 4.4.13 Z-transform filters

Not implemented.

5.2.25 4.5.1 Analysis

Compliant except for “nodeset”

5.2.26 4.5.2 DC analysis

Compliant except for “nodeset”

5.2.27 4.5.4.3 Noise_table

Not implemented

5.2.28 4.6.1 Defining an Analog Function

Compliant except cannot use local parameters

5.2.29 4.6.2 Returning a Value from an Analog Function

Partially compliant. Can use return value for output. Output via passed argument is not supported.

5.2.30 5.3.2 Indirect Branch Assignments

Not implemented

5.2.31 6.7.5. Above Function

Not implemented

5.2.32 7 Hierarchical Structures

In general, hierarchical structures are not supported by the SIMetrix Verilog-A implementation and this is the most siginicant feature omitted at this time. However, much of the functionality provided by this feature may be achieved via the netlist, so this should not impact on the usefulness of the compiler too much. We do intend to implement this in a future version.

This section of the standard does include the syntax for module definitions and this is of course fully supported. This is covered by the opening paragraphs of section 7.1. The rest of the section is not implemented.

5.2.33 8

Not implemented in Verilog-A

5.2.34 9 Scheduling Semantics

Most of this section is concerned with Verilog-AMS which is the mixed-signal version and so is not relevant.

5.2.35 10.1 Environment Parameter Functions

$realtime is not supported. All others are compliant.

5.2.36 10.2 $random Function

Supported for first argument only. 'type_string' argument is not supported.

5.2.37 10.2 $dist_ Functions

Not supported

5.2.38 10.4 Simulation Control System Tasks

Compliant except argument to functions are ignored.

5.2.39 10.7 Announcing Discontinuity

Accepted but doesn't actually do anything

5.2.40 10.9 Limiting Functions

Compliant using built-in "pnjlim". User functions not implemented.

5.2.41 10.10 Hierarchical System Parameter Functions

$mfactor implemented. Others are not

5.2.42 11.1 'default_discipline

Not implemented

5.2.43 11.2 'default_transition

Not implemented

5.2.44 11.6 'resetall

Not implemented

5.2.45 11.7 Pre-defined Macros

Not implemented

5.2.46 12, 13

Not implemented

5.3 SIMetrix Extensions

5.3.1 In an Ideal World...

... any standard would be so carefully designed and thought out that nobody would need to make non-standard extensions. It is our intention to make the SIMetrix Verilog-A implementation follow the standard as closely as possible so that anyone who writes Verilog-A code will be able to use it with another implementation.

While that is our idealistic intention, reality never allows ideals. Verilog-A has quite a few little limitations that we would not want to impose on our users. Some of these we have already addressed and made non-standard extensions to do so. These are detailed below.

We will endeavour in the long run to make such extensions in a manner that would allow a source file to work with other Verilog-A simulators without modification.

5.3.2 Analog Operator Syntax

According to the syntax specification, analog operators, e.g. ddt(), limexp(), white_noise(), may only be used in a standalone manner and may not be embedded in expressions. For example, this is allowed:

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

but this isn't:

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

But you would be allowed to do this:

```
dd = ddt(I(n1,n2)) ;
V(n1, n2) <+ C * dd ;
```

This limitation doesn't make any sense. It might make sense if any variable that an analog operator was assigned to was required to have a discipline defined. Then, the we could make sense of what tolerances to use for ddt() operations for example. But such a requirement is not present, indeed there is no method of assigning a discipline to a variable. So tolerance for ddt() is somewhat hit and miss anyway.

The later language reference manual version 2.3 does in fact not impose the above restrictions. It's possible that these restrictions are a consequence of errors in the definition in the language and not actually intentional.

Currently the $limit function remains subject to the above limitation. But we plan to change this in a future revision.

5.3.3 Instance Parameters

The Verlog-A language does not distinguish between instance parameters and model parameters. An instance parameter is one that can be defined on the device line on a per instance basis whereas a model parameter is one defined in a .MODEL statement. The most flexible implementation is one that allows both, with the instance parameter taking precedence if both are specified by the user. However this method has a cost in terms of increased memory usage per instance. While memory consumption may not seem to be a big issue, it can impact on performance. The less memory used, the more likely that the processor will find what it wants in the cache. For this reason it is desirable to minimise the number of instance parameters.

The SIMetrix Verilog-A implementation provides two methods of defining instance parameters: one in the verilog-A source file and the other on the command line of va.exe which in turn can be passed from .LOAD.

To define an instance parameter in the .VA file, prefix the parameter key word with the special attribute 'type' with a value of "instance". This is how it should look:

```
(* type="instance" *) parameter a = 1 ;
```

To define on .LOAD, add the parameter "instparams=*parameter_list*" where *parameter_list* is a comma separated list of parameter names.

If a parameter is defined as an instance parameter, it will also be available as a model parameter. If both are specified, the instance value will take precedence.

5.3.4 Device Mapping

You may control how the new device is represented in SIMetrix using a device mapping. This does the same as the sxcfg file. Mappings are applied as a module attribute in the form:

(* Mappings="*mapping_defs*" *)

This should prefix the "module" keyword.

mapping_def is a semi-colon delimited list of mapping definitions. Each mapping definition is itself a comma delimited list of attibutes in the following order:

model-type-name,level-number,device-letter,default-parameter,version

Where:

model-type-name	The name used in the .MODEL statement.
level	The LEVEL parameter value in the .MODEL statement.
device-letter	The device letter to use for this device.
default-parameter	A single parameter name and value. This is intended to be used to define device polarity. E.g. "pnp=1" might define a PNP BJT. This is useful to allow the definition of BJTs and MOS devices using conventional NPN/PNP or NMOS/PMOS model type names.
version	Value of VERSION parameter.

For example, the HICUM device is defined with the following mapping:

(* Mappings="hicum_211,0;npn,8,Q,pnp=0,;pnp,8,Q,pnp=1," *)

This has three mappings. You can use hicum_211 with no level parameter to define a model. In this case the pnp parameter would need to be set for a PNP device. Alternatively you can use NPN as a model type name along with LEVEL=8 for an NPN device, or PNP with LEVEL=8 for a PNP device.

5.3.5 Tolerances

The Verilog-A language only allows for absoulte tolerances to be hardwired in the VA source file. This means for example, that absolute current tolerance, must be specified as a fixed constant which cannot be changed in the .OPTIONS line or anywhere else.

SIMetrix provides a workaround for this using the special values $abstol, $vntol, $chgtol and $fluxtol. These can be used to define absolute tolerances in electrical nature definitions. These are already used in the standard discipline header files supplied with the SIMetrix Verilog-A compiler.

5.3.6 Analysis() Function

Additional analyis types:

"sens" sensitivity analysis

"tf" transfer function analysis

"pta" pseudo-transient analysis

"smallsig" small signal analysis

"rtn" real time noise analysis

5.3.7 $simparam() Function

Standard types supported by SIMetrix:

"gdev"

"gmin"

"simulatorSubversion"

"simulatorVersion"

"sourceScaleFactor" - includes pseudo transient scale factor

"tnom"

Additional SIMetrix extensions:

"ptaScaleFactor" - as "sourceScaleFactor" but functional in pseudo transient analysis only. Default = 1.0.

In addition you can specify any option setting defined using .OPTIONS. E.g. $simparam("reltol") will return the value of the RELTOL option.

5.3.8 $fopen() Function

Use the argument "<listfile>" to write to the list file. This is the file created by every simulation with the extension .OUT.

5.4 Verilog-A Interaction with SIMetrix Features

5.4.1 Real-Time Noise

Real-time noise, while not unique to SIMetrix, remains a feature that can only be found on a few simulators. Because of this, standards such as Verilog-A do not account for it or support it in any way. The Verilog-A LRM simply says that transient noise should be implemented by the $random function.

The SIMetrix Verilog-A compiler does fully support the real-time noise feature and the regular small-signal noise analog operators such as white_noise and flicker_noise will correctly create noise signals in transient analysis with real time noise enabled without requiring any special support in the Verilog-A code.

5.4.2 Transient Snapshots

In general it is best to assume that transient snapshots will not work with Verilog-A devices. They will in fact work with some depending on what analog operators and/or system functions are used.

5.4.3 Pseudo-Transient Analysis

Pseudo transient analysis will work correctly with Verilog-A devices provided they are not energy sources. Put another way, if all output sources are zero when all input probes are zero, PTA will work. If there are any sources that are non-zero with zero inputs then PTA performance may be compromised. In this situation you should use the `$simparam("sourceScaleFactor")` system function to scale energy producing outputs. For example a 5V fixed voltage source should look like this:

```
V(n1,n2) <+ 5*$simparam("sourceScaleFactor") ;
```

$simparam("sourceScaleFactor") returns a value from 0.0 to 1.0 representing the supply ramp in pseudo transient as well as DC source stepping.

It would be possible for the compiler to automatically add this. Currently this isn't done as this will not necessarily be beneficial if the device is not energy producing and could lead to a singular matrix condition in some cases. For this reason we currently put the onus on you the user to define PTA behaviour.

Copyright © SIMetrix Technologies Ltd. 1992-2015

SIMetrix 8.0 Verilog A Manual