

# **simetrix**

## **SIMETRIX VERILOG-A MANUAL**

**VERSION 9.2**

OCTOBER 2024

# **SIMETRIX VERILOG-A MANUAL**

**COPYRIGHT © SIMETRIX TECHNOLOGIES LTD. 1992-2024**

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](mailto:support@simetrix.co.uk)

Web: <http://www.simetrix.co.uk>



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	What Is Verilog-A? . . . . .	1
1.2	Verilog-A Language Reference Manual . . . . .	1
<b>2</b>	<b>Using Verilog-A Compiler</b>	<b>2</b>
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	Permanent .SXDEV Installation . . . . .	4
<b>3</b>	<b>Writing Verilog-A Code</b>	<b>5</b>
3.1	Overview . . . . .	5
3.2	Verilog-A Text Editor . . . . .	5
3.3	Hello World! . . . . .	6
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	Pin Location Attribute . . . . .	11
3.6.2	Variables . . . . .	11
3.6.3	analog initial block . . . . .	11
3.6.4	\$fatal . . . . .	12
3.6.5	Functions . . . . .	12
3.6.6	Local Parameters . . . . .	12
3.6.7	Parameter Limits . . . . .	12
3.6.8	Conditional Statements . . . . .	12
3.7	Hysteresis Block . . . . .	13
3.8	A Capacitor . . . . .	13
3.9	A Voltage Controlled Oscillator . . . . .	14
3.10	Digital Elements - Overview . . . . .	15
3.11	Digital Gate . . . . .	15
3.11.1	cross() Monitored Event . . . . .	16
3.11.2	transition() Analog Operator . . . . .	16
3.12	Butterworth Filter . . . . .	17
3.12.1	Arrays . . . . .	18
3.12.2	For Loops . . . . .	19
3.12.3	laplace_nd Function . . . . .	19
3.13	RC Ladder . . . . .	20
3.13.1	Vectors of Nodes . . . . .	20

3.13.2	Analog For Loops and genvars . . . . .	21
3.13.3	Compile-time Parameters . . . . .	21
3.13.4	See Also . . . . .	21
3.14	Indirect Assignments . . . . .	21
3.14.1	Virtual Earth . . . . .	22
3.14.2	Differential Equation . . . . .	23
3.15	Phase-locked Loop . . . . .	24
3.15.1	Top-level Module . . . . .	24
3.15.2	Sub-module Definitions . . . . .	25
3.15.3	Hierarchical Structures . . . . .	26
<b>4</b>	<b>Verilog-A Reference</b> . . . . .	<b>27</b>
4.1	Verilog-A Functions . . . . .	27
4.1.1	\$abstime . . . . .	30
4.1.2	\$arandom . . . . .	30
4.1.3	\$bound_step . . . . .	31
4.1.4	\$clog2 . . . . .	32
4.1.5	\$debug . . . . .	32
4.1.6	\$discontinuity . . . . .	32
4.1.7	\$display . . . . .	32
4.1.8	\$dist_chi_square . . . . .	34
4.1.9	\$dist_erlang . . . . .	34
4.1.10	\$dist_exponential . . . . .	35
4.1.11	\$dist_normal . . . . .	35
4.1.12	\$dist_poisson . . . . .	35
4.1.13	\$dist_t . . . . .	35
4.1.14	\$dist_uniform . . . . .	35
4.1.15	\$error . . . . .	35
4.1.16	\$fatal . . . . .	36
4.1.17	\$fclose . . . . .	36
4.1.18	\$fdebug . . . . .	37
4.1.19	\$fdisplay . . . . .	37
4.1.20	\$ferror . . . . .	37
4.1.21	\$fflush . . . . .	38
4.1.22	\$fgets . . . . .	38
4.1.23	\$finish . . . . .	38
4.1.24	\$fmonitor . . . . .	39
4.1.25	\$fopen . . . . .	39
4.1.26	\$fscanf . . . . .	40
4.1.27	\$fseek . . . . .	40
4.1.28	\$fstrobe . . . . .	40
4.1.29	\$ftell . . . . .	41
4.1.30	\$fwrite . . . . .	41
4.1.31	\$info . . . . .	41
4.1.32	\$mfactor . . . . .	42
4.1.33	\$monitor . . . . .	42
4.1.34	\$param_given . . . . .	42
4.1.35	\$port_connected . . . . .	42
4.1.36	\$random . . . . .	43
4.1.37	\$rdist_chi_square . . . . .	43
4.1.38	\$rdist_erlang . . . . .	44
4.1.39	\$rdist_exponential . . . . .	44
4.1.40	\$rdist_normal . . . . .	44
4.1.41	\$rdist_poisson . . . . .	44
4.1.42	\$rdist_t . . . . .	44
4.1.43	\$rdist_uniform . . . . .	44

4.1.44	\$rewind	44
4.1.45	\$sformat	44
4.1.46	\$sscanf	45
4.1.47	\$simparam	46
4.1.48	\$simparam\$str	46
4.1.49	\$simprobe	47
4.1.50	\$stop	47
4.1.51	\$strobe	48
4.1.52	\$swrite	48
4.1.53	\$table_model	48
4.1.54	\$temperature	53
4.1.55	\$vt	53
4.1.56	\$warning	53
4.1.57	\$write	54
4.1.58	above	54
4.1.59	abs	54
4.1.60	absdelay	54
4.1.61	ac_stim	55
4.1.62	acos	55
4.1.63	acosh	55
4.1.64	analysis	55
4.1.65	asin	56
4.1.66	asinh	56
4.1.67	atan	56
4.1.68	atan2	56
4.1.69	atanh	56
4.1.70	ceil	56
4.1.71	cos	56
4.1.72	cosh	56
4.1.73	cross	57
4.1.74	ddt	58
4.1.75	ddx	58
4.1.76	exp	58
4.1.77	flicker_noise	59
4.1.78	floor	59
4.1.79	hypot	59
4.1.80	idt	59
4.1.81	idtmod	60
4.1.82	laplace_nd	60
4.1.83	laplace_np	61
4.1.84	laplace_zd	61
4.1.85	laplace_zp	61
4.1.86	last_crossing	62
4.1.87	limexp	62
4.1.88	ln	63
4.1.89	log	63
4.1.90	max	63
4.1.91	min	63
4.1.92	pow	63
4.1.93	sin	63
4.1.94	sinh	63
4.1.95	slew	63
4.1.96	sqrt	64
4.1.97	tan	64
4.1.98	tanh	64
4.1.99	timer	64

4.1.100 transition . . . . .	65
4.1.101 white_noise . . . . .	66
4.1.102 zi_nd . . . . .	66
4.1.103 zi_np . . . . .	67
4.1.104 zi_zd . . . . .	67
4.1.105 zi_zp . . . . .	68
4.2 Analog Operator Restrictions . . . . .	69
<b>5 Implementation - vs LRM</b> . . . . .	<b>71</b>
5.1 Overview . . . . .	71
5.2 SIMetrix Verilog-A vs LRM 2.4 . . . . .	71
5.2.1 2.6 Numbers . . . . .	71
5.2.2 2.7 String Literals . . . . .	71
5.2.3 2.8 Identifiers, Keywords, and System Names . . . . .	72
5.2.4 2.9 Attributes . . . . .	72
5.2.5 2.9.2 Standard Attributes . . . . .	72
5.2.6 3.4.2 Parameters - Value Range Specification . . . . .	72
5.2.7 3.4.3 Parameter Units and Descriptions . . . . .	72
5.2.8 3.6.2.2 Domain Binding . . . . .	72
5.2.9 3.6.2.3 Natureless Disciplines and Domainless Disciplines . . . . .	73
5.2.10 3.6.2.4 Disciplines of Nets and Undeclared Nets . . . . .	73
5.2.11 3.6.2.7 User Defined Attributes . . . . .	73
5.2.12 3.6.3.1 Net Descriptions . . . . .	73
5.2.13 3.6.3.2 Net Discipline Initial (Nodeset) Values . . . . .	73
5.2.14 3.6.5 Implicit Nets . . . . .	73
5.2.15 3.7 Real Net Declarations . . . . .	73
5.2.16 3.8 Default Discipline . . . . .	73
5.2.17 3.9 Disciplines of primitives . . . . .	73
5.2.18 3.10 Discipline Precedence . . . . .	73
5.2.19 3.11 Net compatibility . . . . .	73
5.2.20 3.12 Branches . . . . .	74
5.2.21 4.2.6 Case Equality Operator . . . . .	74
5.2.22 4.2.13 Concatenations . . . . .	74
5.2.23 4.2.14 Assignment Patterns . . . . .	74
5.2.24 4.3 Built-in Mathematical Functions . . . . .	74
5.2.25 4.5.15 Restrictions on Analog Operators . . . . .	74
5.2.26 4.5.3 Time derivative Operator . . . . .	74
5.2.27 4.5.4 Time integral operator . . . . .	74
5.2.28 4.5.5 Circular Integral Operators . . . . .	74
5.2.29 4.6.1 Analysis . . . . .	75
5.2.30 4.6.2 DC analysis . . . . .	75
5.2.31 4.6.4.3 noise_table . . . . .	75
5.2.32 4.6.4.4 noise_table_log . . . . .	75
5.2.33 4.7.1 Defining an Analog Function . . . . .	75
5.2.34 4.7.2 Returning a Value from an Analog Function . . . . .	75
5.2.35 5.10.3.4 absdelta Function . . . . .	75
5.2.36 6 Hierarchical Structures . . . . .	75
5.2.37 7 Mixed Signal . . . . .	75
5.2.38 8 Scheduling Semantics . . . . .	75
5.2.39 9.6 Timescale system tasks . . . . .	76
5.2.40 9.7.1 \$finish System Task . . . . .	76
5.2.41 9.7.2 \$stop System Task . . . . .	76
5.2.42 9.8 . . . . .	76
5.2.43 9.9 . . . . .	76
5.2.44 9.10 Simulator time system functions . . . . .	76
5.2.45 9.11 . . . . .	76

5.2.46	9.12	76
5.2.47	9.13.1 \$random and \$arandom Functions	76
5.2.48	9.13.2 Distribution Functions	76
5.2.49	9.15 Analog kernel parameter system functions	76
5.2.50	9.17.1 \$discontinuity	77
5.2.51	9.17.3 \$limit	77
5.2.52	9.18 Hierarchical System Parameter Functions	77
5.2.53	9.19 Explicit binding detection system functions	77
5.2.54	9.20 Analog node alias system functions	77
5.2.55	9.22	77
5.2.56	9.23	77
5.2.57	10.1 Compiler directives	77
5.2.58	10.5 Predefined Macros	77
5.2.59	11, 12	78
5.2.60	Annex E	78
<b>5.3</b>	<b>SIMetrix Extensions</b>	<b>78</b>
5.3.1	In an Ideal World...	78
5.3.2	Instance Parameters	78
5.3.3	Boolean Parameters	79
5.3.4	SPICE compatibility	79
5.3.5	Output Variables	79
5.3.6	Device Mapping	80
5.3.7	Device Binning	81
5.3.8	Schematic Symbol	81
5.3.9	Tolerances	84
5.3.10	Analysis() Function	84
5.3.11	\$simparam() Function	84
5.3.12	\$fopen() Function	85
5.3.13	Special Parameters	85
<b>5.4</b>	<b>Verilog-A Interaction</b>	<b>85</b>
5.4.1	Real-Time Noise	85
5.4.2	Transient Snapshots	85
5.4.3	Pseudo-Transient Analysis	85
<b>6</b>	<b>Debugging with Microsoft Visual Studio</b>	<b>87</b>
6.1	Introduction	87
6.2	Installing Visual Studio	87
6.2.1	Installing Instruction for Visual Studio 2022	87
6.2.2	Installation Instructions for Visual Studio 2019 and 2017	88
6.3	Checking Compiler	88
6.4	Manual Configuration	89
6.5	Preparation for Debug	89
6.6	Running a Debug Session	89
6.7	Recompiling after Debugging	92
6.8	Debugging Methods	92
6.8.1	Execution Flow	92
6.8.2	C vs Verilog-A	92
6.8.3	Examining Variables	94
6.8.4	Partial Derivatives	94
6.8.5	Pre-evaluated Variables	95
6.8.6	Optimised Variables	95
6.8.7	Multi-thread Execution	95
6.8.8	Extended and Quad Precision	95
6.9	Trademarks	96

# 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 SIMetrix implementation was originally developed based on the version 2.2 Language Reference Manual. From version 8.3 we have updated the documentation and compiler messages to refer to LRM 2.4 which is the latest version. In most areas, LRM 2.4 is a superset of LRM 2.2 and in nearly all cases Verilog-A modules written for LRM 2.2 will work identically in a 2.4 implementation without requiring modification.

The version 2.4 language reference manuals may be obtained from *Verilog-A Language Reference Manual*



## 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 %APPDATA%\SIMetrix\v\v\vacache where %APPDATA% is your application data directory. (v\v is the product version, e.g. 830 for version 8.3)

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. We recommend you avoid using 'Q'; this device can 3 or 4 terminals which can lead to some ambiguities.

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 Verilog-A compiler and the MAKE utility.

In rare cases, there may be warnings or errors generated by the 'C' compiler or linker. These should be reported to technical support.

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 second 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 file or 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](#) 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=011* is an advanced feature for debugging purposes. It tells the compiler to use meaningful names in the 'C' file if possible. Otherwise it will use short names. There is a very 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 Permanent .SXDEV Installation

The .SXDEV files may be relocated to the plugins\devices directory in which case they become a built-in device. Although binary compatibility between versions is not guaranteed, .sxdev files created using an older SIMetrix version will nearly always work correctly with a newer SIMetrix version. 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
<a href="#">Hello World!</a>	Examples/Verilog-A/Manual/Hello-world
<a href="#">A Simple Device Model</a>	Examples/Verilog-A/Manual/Gain-block
<a href="#">A Resistor</a>	Examples/Verilog-A/Manual/Resistor
<a href="#">A Soft Limiter</a>	Examples/Verilog-A/Manual/Soft-limiter
<a href="#">Hysteresis Block</a>	Examples/Verilog-A/Manual/Hysteresis-block
<a href="#">A Capacitor</a>	Examples/Verilog-A/Manual/Capacitor
<a href="#">A Voltage Controlled Oscillator</a>	Examples/Verilog-A/Manual/Vco
<a href="#">Digital Gate</a>	Examples/Verilog-A/Manual/Gates
<a href="#">Butterworth Filter</a>	Examples/Verilog-A/Manual/Butterworth-filter
<a href="#">RC Ladder - Loops, Vectored Nodes and genvars</a>	Examples/Verilog-A/Manual/RC-ladder
<a href="#">Indirect Assignments</a>	Examples/Verilog-A/Manual/Indirect-assignment

### 3.2 Verilog-A Text Editor

SIMatrix 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. 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. You will see a message box pop up advising about a feature that allows the symbol to be defined in the Verilog-A module. We will show you this feature in a later example. Check the Don’t show this message again box the Close.
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 see 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 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 formats 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:

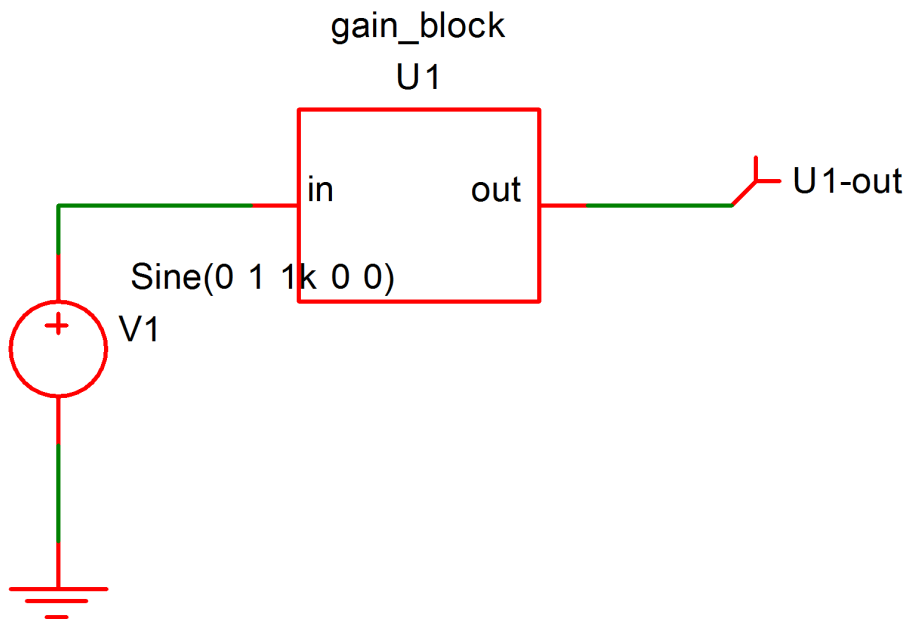
```
`include "disciplines.vams"
module gain_block(in, out) ;

    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 an 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, select the Edit Model Parameters option, 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"

(* symbol="res" *)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"

(* symbol="res" *)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
```



```

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.

Finally the module statement is prefixed with the text:

```
(* symbol="res" *)
```

This is known as an *attribute*. Verilog-A attributes are name-value pairs that can be specific to an implementation. The *symbol* attribute is specific to SIMetrix and defines the library symbol to be used with the module. "res" is the symbol for a box-shape resistor. If you prefer a z-shaped resistor, you can change this to "resz". For more information see [Specify Library Symbol](#). When you select the menu **Verilog-A | Construct Verilog-A Symbol**, the resistor symbol will be offered instead of the generic auto-generated symbol.

## 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"
(* pins="L;R" *)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 initial
        if (vhigh<vlow)
            $fatal(1, "Lower limit must be less than higher limit") ;

    analog
    begin

        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. Pin location attribute. The module attribute (`* pins="L;R"*`) defines the location of the pins for an auto-generated symbol
2. 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
3. The `analog initial` statement
4. The `$fatal` system task. This aborts the simulation immediately and displays the given message
5. The `exp` function
6. Local parameters using the `localparam` keyword
7. 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 Pin Location Attribute

The module attribute (`* pins="L;R"*`) defines the location of the pins for an auto-generated symbol. The value is a list of semi-colon delimited letters that define the location of the corresponding module port pin on the schematic symbol. The letters accepted are L, R, T and B for "Left", "Right", "Top" and "Bottom" respectively. For more information see [Specify Pin Positions](#)

### 3.6.2 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. Alternative declarations are `integer`, which means the variable stores whole numbers, and `string` which defines a variable that takes text strings. 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.

### 3.6.3 analog initial block

This is not the same as the `initial_step` event introduced earlier although in many cases the end result is the same. `analog initial` blocks are executed just once before any simulation starts. `initial_step` event code is executed for every iteration performed during the DC operating point calculation. In most situations any kind of initialisation is most efficiently performed in an `analog initial` block.

Not all Verilog-A statements are allowed in an `analog initial` block.

The following are not permitted.

Expressions containing access functions such as `v(in)`

Expressions containing analog operators such as `ddt`. (These are introduced in later examples)

Branch contributions

Event statements (introduced in later examples)

### 3.6.4 \$fatal

The `$fatal` system task aborts the simulation unconditionally and immediately and displays the given message. For more information see [\\$fatal](#)

### 3.6.5 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](#) for a complete list.

### 3.6.6 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 values as, unlike variables, they cannot be assigned once 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 cannot be subsequently modified. This aids readability but more importantly tells the compiler that it cannot change allowing it to optimise the result effectively.

### 3.6.7 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]
```

### 3.6.8 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 Hysteresis Block

This is a simple design that demonstrates using state variables. The clock has a single input and a single output. When the input exceeds 3V the output switches to a high state. When the input falls below 2V the output switches to a low state. As the state of the switch cannot be unambiguously determined from the input, the Verilog-A code uses an integer variable to store the state of the switch.

```
`include "disciplines.vams"

module hysteresis_block(vc, out) ;

    electrical vc, out ;

    integer state ;
    real vout ;

    parameter real HIGH=5.0 ;
    parameter real LOW= 0.0 ;

    analog initial
        state = 1 ;

    analog
    begin

        @(cross(V(vc)-3,1))
            state=1 ;
        @(cross(V(vc)-2,-1))
            state=0 ;

        if (state==1)
            vout = HIGH ;
        else
            vout = LOW ;

        V(out) <+ transition(vout, 0, 1n, 1n) ;
    end

endmodule
```

The state variable in the above is the integer `state`. There is nothing in the code that declares the variable as a state variable; the Verilog-A compiler detects this automatically. The two `cross` event statements set `state` when the input passes the desired thresholds. The analog initial block is used to set the starting state. For simplicity, this is set to 1.

## 3.8 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"

(* symbol="cap_simple_subckt" *)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.

## 3.9 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.

## 3.10 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.

## 3.11 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

### 3.11.1 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))
```

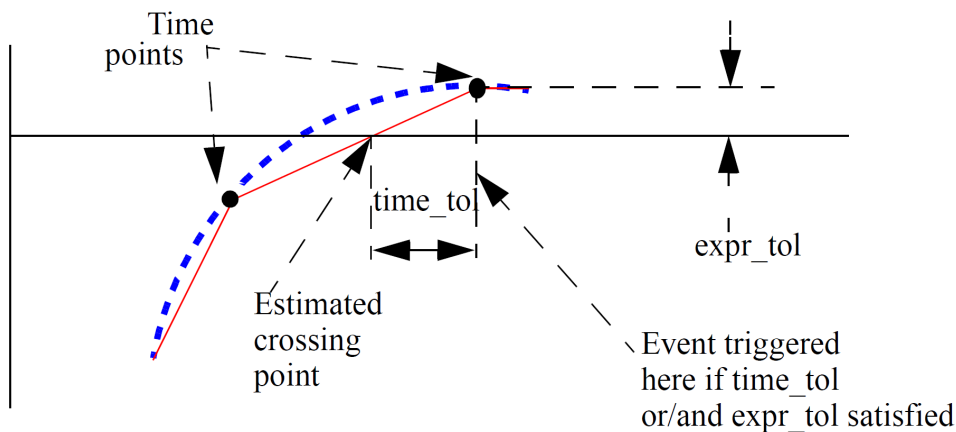
This line both defines the event and also responds to the event when it is triggered. The arguments define the event, while the statement that follows it is the action taken when the event is triggered.

The function has the following form:

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

Only the first argument is compulsory.

<i>expr</i>	expression to test. The event is triggered when the expression crosses zero.
<i>edge</i>	0, +1 or -1 to indicate edge. +1 means the event will only occur when <i>expr</i> is rising, -1 means it will only occur while falling and 0 means it will occur on either edge. Default=0 if omitted.
<i>time_tol</i>	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. <i>time_tol</i> 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 <i>time_tol</i> . 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.
<i>expr_tol</i>	Similar to <i>time_tol</i> but instead defines the tolerance on the input expression. See dialog below.



Cross Event Function Behaviour

### 3.11.2 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

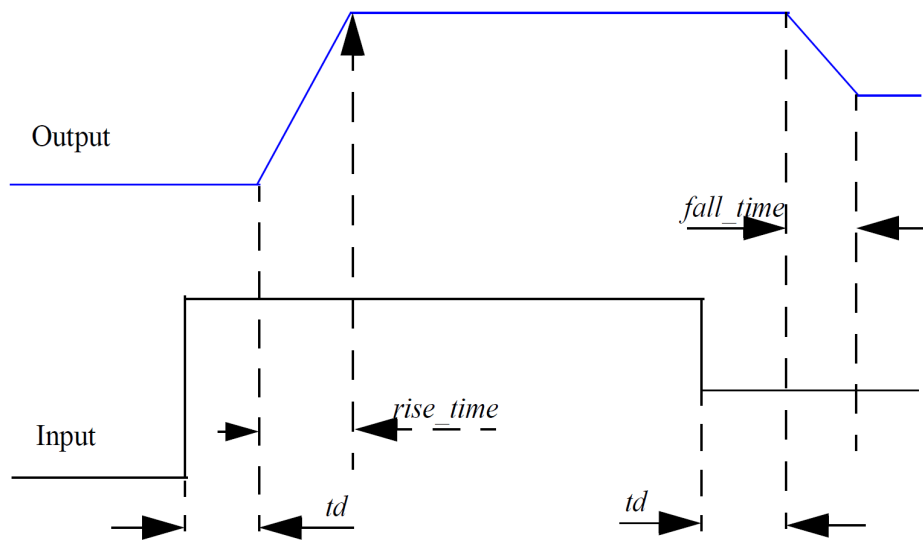
the and gate example above, the output logic level can change instantaneously but the output of a real device would typically follow a specified rise or fall time. The transition analog operator converts the discrete input value to a continuous output value using specified rise and fall times. The function has the following form:

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

<code>expr</code>	Input expression.
<code>td</code>	Delay time. This is a <i>transport</i> or <i>stored</i> 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 <i>inertial</i> delay which swallows activity that has a shorter duration than the delay. Default=0.
<code>rise_time</code>	Rise time of output in response to change in input.
<code>fall_time</code>	Fall time of output in response to change in input.
<code>time_tol</code>	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 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

## 3.12 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 initial
    begin
        scale = 1.0/freq/2/`M_PI ;
        bPrev = 1.0 ;
        den[0] = 1.0 ;

        //      Calculate Butterworth coefficients
        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 coef %d = %g", k, den[k]) ;
        end
    end

    analog
    begin
        // 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 `laplace_nd` analog operator

### 3.12.1 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:

<i>type</i>	real or integer
<i>array_name</i>	name of array
<i>low_index</i>	Minimum index allowed
<i>high_index</i>	Maximum index allowed

*low\_index* and *high\_index* determine the number of elements in the array to be  $high\_index - low\_index + 1$ .

### 3.12.2 For Loops

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

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

<i>initial_assignment</i>	Assignment statement (in the form <i>variable = expression</i> ) 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.
<i>test_expression</i>	Expression is evaluated at the start of each iteration around the loop before <i>statement</i> . If the result of the evaluation is non-zero, <i>statement</i> will be executed. If not the loop will be terminated.
<i>loop_assignment</i>	Assignment statement that is executed after <i>statement</i> . Typically this would be an assignment that increments or decrements a loop counter variable. In the above it increments k by 1.

### 3.12.3 laplace\_nd Function

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

$$H(s) = \frac{n_0 + n_1s + n_2s^2 + \dots + n_ms^m}{d_0 + d_1s + d_2s^2 + \dots + d_ms^m}$$

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

The `laplace_nd` function is in the form:

```
laplace_nd (expr, num_coeffs, den_coeffs,  $\epsilon$ )
```

Where

<i>expr</i>	Input expression
<i>num_coeffs</i>	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.
<i>den_coeffs</i>	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.
$\epsilon$	Tolerance parameter currently unused.

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.

### 3.13 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
    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)

#### 3.13.1 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.

### 3.13.2 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.

### 3.13.3 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.

### 3.13.4 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.

## 3.14 Indirect Assignments

Indirect assignments are a way of defining an equation to be solved. Here we present two examples of indirect assignments, a perfect virtual earth and the solution of a first order differential equation.

### 3.14.1 Virtual Earth

Consider this example of an indirect branch assignment:

```
V(out) : V(in) == 0.0 ;
```

This can be expressed in words as "drive V(out) so that V(in) equals zero". This creates a virtual earth at vin.

Here is the complete Verilog-A module:

```
`include "disciplines.vams"

module virtual_earth(in, out) ;

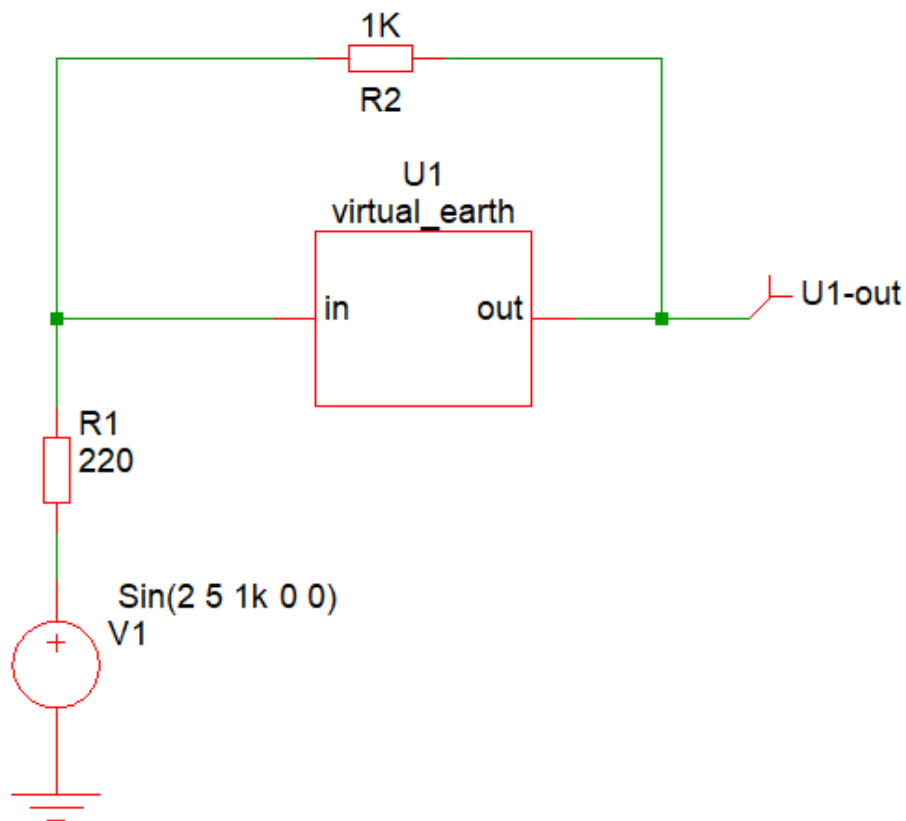
    electrical in, out;

    analog
    begin

        V(out) : V(in) == 0.0 ;

    end
endmodule
```

The above can be connected in the following circuit:



In the above circuit, the *in* node of U1 will be maintained at exactly zero volts therefore fabricating an ideal inverting amplifier with a gain of  $1k/220$ .

### 3.14.2 Differential Equation

Consider this Verilog-A module:

```
`include "disciplines.vams"

module differential_equation(in, out) ;

    electrical in, out;

    analog
    begin

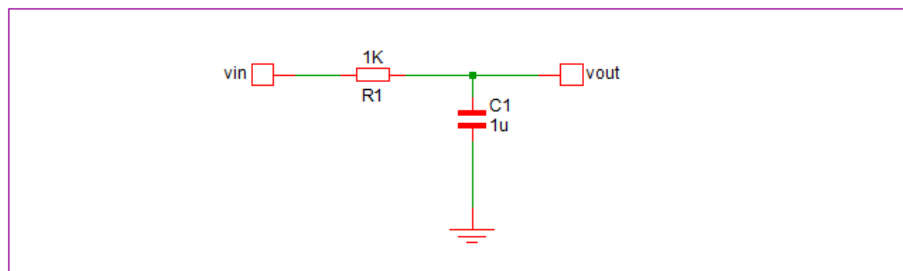
        V(out) : ddt(V(out)) == (V(in) - V(out))/1m ;

    end
endmodule
```

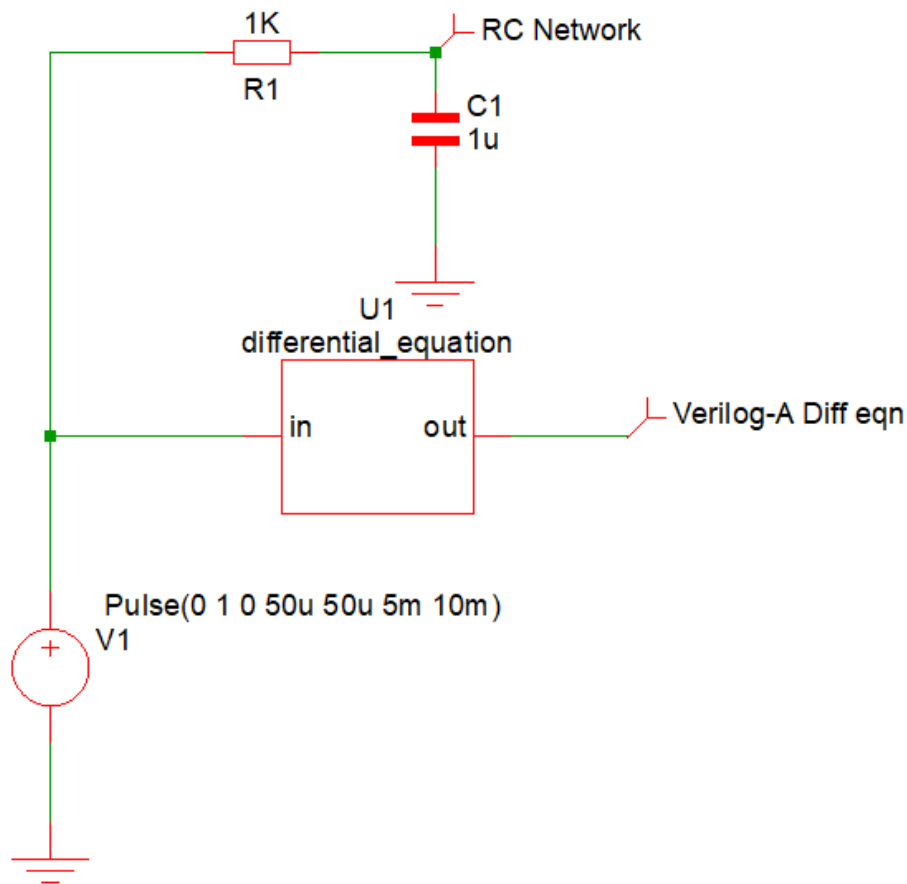
This solves the differential equation:

$$\frac{d.vout}{dt} = \frac{(vin - vout)}{0.001}$$

This is the equation that governs the following circuit:



You can try this with this schematic:



## 3.15 Phase-locked Loop

In this example, a Phase-locked loop is described. The PLL comprises three blocks each of which is implemented with a single Verilog-A module. The three modules are then interconnected using the hierarchical structures feature of Verilog-A

Hierarchical structures provide a means of interconnecting Verilog-A modules or SIMetrix primitives within a Verilog-A module. This makes it possible to construct higher level modules to implement a subsystem or even a complete circuit.

### 3.15.1 Top-level Module

```

module pll(vin_rf, vlocal_osc, phase_detect_in, vout);

    electrical vin_rf, vlocal_osc, vout_ph_det, vout, phase_detect_in;

    parameter real FILTER_BANDWIDTH=130.0;
    parameter real VCO_GAIN=1K;
    parameter real VC_CENTRE_FREQ=2.45K;

    phase_detector #(.GAIN(2))
        pd(phase_detect_in, vin_rf, vout_ph_det);

```

```

vco #(.AMPLITUDE(1),.CENTRE_FREQ(VCO_CENTRE_FREQ), .VCO_GAIN(VCO_GAIN) )
    osc (vout, vlocal_osc);

lpf_1storder #(.BANDWIDTH(FILTER_BANDWIDTH) )
    lpf (vout_ph_det, vout);

endmodule

```

The above instantiates three sub-modules, that is the phase detector, VCO and low-pass filter. For example in:

```

phase_detector #(.GAIN(2))
    pd (phase_detect_in, vin_rf, vout_ph_det);

```

`phase_detector` specifies the module name.

`pd` is the name of the instance and is arbitrary but must be unique.

The arguments to `#( )` are the parameter values. So `.GAIN(2)` sets the `GAIN` parameter to 2.

`phase_detect_in`, `vin_rf`, `vout_ph_det` define the connections to the module.

### 3.15.2 Sub-module Definitions

```

// Simple first order filter
module lpf_1storder(vin, vout);

    electrical vout, vin;

    parameter real BANDWIDTH = 300.0;

    localparam r = 1k ;
    localparam c= 1.0/(2.0*`M_PI*r*BANDWIDTH);

    analog begin
        V(vout, vin) <+ r*I(vout, vin);
        I(vout) <+ ddt(c*V(vout));
    end

endmodule

// Ideal VCO
module vco(vin, vout);

    electrical vin, vout;

    parameter real AMPLITUDE = 1.0;
    parameter real CENTRE_FREQ = 2.5k;
    parameter real VCO_GAIN = 1K;

    real phase;
    real inst_freq;

    analog begin

        inst_freq = CENTRE_FREQ + VCO_GAIN * V(vin);
        phase = 2.0 * `M_PI * idtmod( inst_freq, 0.0, 1.0);

        V(vout) <+ AMPLITUDE * sin ( phase);

        $bound_step (0.04 / inst_freq);
    end
endmodule

```



```
// Phase detector - using multiplier
module phase_detector(vlocal_osc, vin_rf, vif);

    electrical vlocal_osc, vin_rf, vif;
    parameter real GAIN = 2.0;

    analog begin
        V(vif) <+ GAIN*V(vlocal_osc)*V(vin_rf);
    end

endmodule
```

### 3.15.3 Hierarchical Structures

Hierarchical structures are a Verilog concept that allow interconnection of modules at different levels of hierarchy. This is essentially the same function as the netlist but defined within a Verilog-A module.

In this phase-locked loop example, all modules and the top level module interconnecting them are in the same file. However this isn't necessary and it is possible to have each Verilog-A module in its own file. In this case a .LOAD statement must be included in the netlist to load all Verilog-A module accessed in the instance statements.

# Chapter 4

## Verilog-A Reference

The official definition of the Verilog-A language can be found in the Language Reference Manual (LRM) 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?
<a href="#">\$abstime</a>	real	()	Yes
<a href="#">\$angle</a>	real	()	No
<a href="#">\$arandom</a>	integer	(integer,[string])	Yes
<a href="#">\$bound_step</a>	none	(real)	Yes
<a href="#">\$debug</a>	none	([real/integer/string...])	Yes
<a href="#">\$discontinuity</a>	none	([integer])	Yes
<a href="#">\$display</a>	none	([real/integer/string...])	Yes
<a href="#">\$dist_chi_square</a>	integer	(integer,integer)	Yes
<a href="#">\$dist_erlang</a>	integer	(integer,integer,integer)	Yes
<a href="#">\$dist_exponential</a>	integer	(integer,integer)	Yes
<a href="#">\$dist_normal</a>	integer	(integer,integer,integer)	Yes
<a href="#">\$dist_poisson</a>	integer	(integer,integer)	Yes
<a href="#">\$dist_t</a>	integer	(integer,integer)	Yes
<a href="#">\$dist_uniform</a>	integer	(integer,integer,integer)	Yes
<a href="#">\$error</a>	none	([string...])	Yes
<a href="#">\$fatal</a>	none	(integer, [string...])	Yes
<a href="#">\$fclose</a>	none	(integer)	Yes
<a href="#">\$fdebug</a>	none	(integer, [real/integer/string...])	Yes
<a href="#">\$fdisplay</a>	none	(integer, [real/integer/string...])	Yes
<a href="#">\$ferror</a>	integer	(integer, string)	Yes
<a href="#">\$fflush</a>	none	(integer)	Yes

Name	Return type	In types	Implemented?
\$fgets	integer	(string, integer)	Yes
\$finish	none	([integer])	Yes
\$fmonitor	none	(integer, [real/integer/string...])	Yes
\$fopen	integer	(string, [string])	Yes
\$fscanf	integer	(integer, string [real/integer/string...])	Yes
\$fseek	integer	(integer, integer, integer)	Yes
\$fstrobe	none	(integer, [real/integer/string...])	Yes
\$ftell	integer	(integer)	Yes
\$fwrite	none	(integer, [real/integer/string...])	Yes
\$info	none	([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)	Yes
\$rdist_chi_square	real	(integer,real, [string])	Yes
\$rdist_erlang	real	(integer,real,real, [string])	Yes
\$rdist_exponential	real	(integer,real, [string])	Yes
\$rdist_normal	real	(integer,real,real, [string])	Yes
\$rdist_poisson	real	(integer,real, [string])	Yes
\$rdist_t	real	(integer,real, [string])	Yes
\$rdist_uniform	real	(integer,real,real,[string])	Yes
\$realtime	real	([real])	No
\$rewind	none	(integer)	Yes
\$sformat	none	(string, string [real/integer/string...])	Yes
\$sscanf	integer	(string, string [real/integer/string...])	Yes
\$simparam	real	(string,[real])	Yes
\$stop	none	([integer])	Yes
\$strobe	none	([real/integer/string...])	Yes
\$swrite	none	(string, string [real/integer/string...])	Yes
\$stable_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])	Yes

Name	Return type	In types	Implemented?
<a href="#">abs</a>	copies args	(real/int)	Yes
<a href="#">absdelay</a>	real	(real, real, [real])	Yes
<a href="#">ac_stim</a>	complex	(string,[real,[real]])	Yes
<a href="#">acos</a>	real	(real)	Yes
<a href="#">acosh</a>	real	(real)	Yes
<a href="#">analysis</a>	integer	(string,...)	Yes
<a href="#">asin</a>	real	(real)	Yes
<a href="#">asinh</a>	real	(real)	Yes
<a href="#">atan</a>	real	(real)	Yes
<a href="#">atan2</a>	real	(real,real)	Yes
<a href="#">atanh</a>	real	(real)	Yes
<a href="#">ceil</a>	real	(real)	Yes
<a href="#">cos</a>	real	(real)	Yes
<a href="#">cosh</a>	real	(real)	Yes
<a href="#">cross</a>	integer	(real,[integer,[real,[real]]])	Yes
<a href="#">ddt</a>	real	(real,[real/string])	Yes
<a href="#">ddx</a>	real	(real,access_func)	Yes
<a href="#">exp</a>	real	(real)	Yes
<a href="#">flicker_noise</a>	real	(real,real,[string])	Yes
<a href="#">floor</a>	real	(real)	Yes
<a href="#">hypot</a>	real	(real,real)	Yes
<a href="#">idt</a>	real	(real,[real,[real,[real/string]]])	Yes
<a href="#">idtmod</a>	real	(real,[real,[real,[real,[real/string]]]])	Yes
<a href="#">laplace_nd</a>	real	(real,real-array,real-array,[real/string])	Yes
<a href="#">laplace_np</a>	real	(real,real-array,real-array,[real/string])	Yes
<a href="#">laplace_zd</a>	real	(real,real-array,real-array,[real/string])	Yes
<a href="#">laplace_zp</a>	real	(real,real-array,real-array,[real/string])	Yes
<a href="#">last_crossing</a>	real	(real,integer)	Yes
<a href="#">limexp</a>	real	(real)	Yes
<a href="#">ln</a>	real	(real)	Yes
<a href="#">log</a>	real	(real)	Yes
<a href="#">max</a>	copies args	(real/int,real/int)	Yes
<a href="#">min</a>	copies args	(real/int,real/int)	Yes
<a href="#">noise_table</a>	real	(real-array,[string])	No
<a href="#">pow</a>	real	(real,real)	Yes
<a href="#">sin</a>	real	(real)	Yes
<a href="#">sinh</a>	real	(real)	Yes
<a href="#">slew</a>	real	(real,[real,[real]])	Yes
<a href="#">sqrt</a>	real	(real)	Yes
<a href="#">tan</a>	real	(real)	Yes

Name	Return type	In types	Implemented?
<a href="#">tanh</a>	real	(real)	Yes
<a href="#">timer</a>	integer	(real,[real,[real]])	Yes
<a href="#">transition</a>	real	(real,[real,[real,[real,[real]]]])	Yes
<a href="#">white_noise</a>	real	(real,[string])	Yes
<a href="#">zi_nd</a>	real	(real,real-array,real-array,real,[real,[real]])	Yes
<a href="#">zi_np</a>	real	(real,real-array,real-array,real,[real,[real]])	Yes
<a href="#">zi_zd</a>	real	(real,real-array,real-array,real,[real,[real]])	Yes
<a href="#">zi_zp</a>	real	(real,real-array,real-array,real,[real,[real]])	Yes

### 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 \$arandom

```
value = $arandom([seed], [type_string]) ;
```

Where *seed* is an integer and *type\_string* is a string with value "global" or "instance".

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

#### Mode 1: no seed

\$arandom 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 \$arandom will return a new random number on each call using the supplied seed for the starting value. In this mode \$arandom will always return the same sequence of values for a given seed.

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 be repeatable given the same initial value for *seed*.

Example:

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

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

### type\_string argument

The *type\_string* argument may be used when *\$random* is used in an expression which initialises a parameter. If set to "global", all calls to *\$random* with a given seed will return the same random sequence. If set to "instance", the random sequence will vary according to the instance that calls it. All calls for a given instance will return the same sequence.

Note that the use of the *type\_string* argument in a parameter initialisation expression is non-standard and may trigger an error if used with other Verilog-A implementations. LRM 2.4 states that *type\_string* may only be specified in a *paramset* statement. SIMetrix does not currently implement *paramset*; these are used for hierarchical construction of Verilog-A modules.

SIMetrix also allows *type\_string* to be used in an analog block expression, but a warning message will be displayed if this is used.

Finally, an "instance" *type\_string* will only have an effect if specified for an instance parameter. Instance parameters are specified using a special attribute. E.g.:

```
(* type="instance" *)parameter integer ran_num = $random(1, "instance") ;
```

### See Also

[\\$random](#)

[\\$rdist\\_chi\\_square](#)

[\\$rdist\\_erlang](#)

[\\$rdist\\_exponential](#)

[\\$rdist\\_normal](#)

[\\$rdist\\_poisson](#)

[\\$rdist\\_t](#)

[\\$rdist\\_uniform](#)

#### 4.1.3 \$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.

#### 4.1.4 \$clog2

```
$clog2( unsigned_integer_argument ) ;
```

Returns  $\text{ceil}(\log_2(\text{argument}+1))$ . It is the minimum number of bits required to represent the integer argument. Note that negative numbers are treated as 32 bit unsigned. E.g. -1 becomes 4294967295 and thus  $\$clog2(-1) = 32$ .

#### 4.1.5 \$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.

\$debug writes an implicit new line character at the end of the text.

#### See Also

[\\$fdebug](#)

[\\$display](#)

#### 4.1.6 \$discontinuity

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

Does not return a value.

Currently \$discontinuity performs no action.

#### 4.1.7 \$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 <a href="#">Real Number Formats</a> below
%f or %F	Real number format. See <a href="#">Real Number Formats</a> below
%g or %G	Real number format. See <a href="#">Real Number Formats</a> below
%r or %R	Real number format. See <a href="#">Real Number Formats</a> 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

The compiler will raise an error if there is a string/numeric mismatch between the format string and supplied arguments. That is if %s found in the format corresponds to an integer or real argument an error will be raised. Conversely if %d or %g etc found in the format string corresponds to a string variable an error will also be raised.

For example, the following will raise an error:

```
$display("Count=%g", "hello world") ;
```

No error will result from a real/integer mismatch but a warning will be raised if the format string specifies an integer type but the supplied value is a real.

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

[\\$display](#)

[\\$debug](#)

[\\$monitor](#)

### 4.1.8 \$dist\_chi\_square

```
integer_value = $rdist_chi_square( seed, degrees_of_freedom) ;
```

This function returns a random number with a chi square statistical distribution. *seed* must be an integer variable initialised with the initial seed value. Each call to the function will update the seed value.

All arguments and return values are integers. In most analog applications the real-valued version [\\$rdist\\_chi\\_square](#) is more appropriate.

### 4.1.9 \$dist\_erlang

```
integer_value = $rdist_erlang( seed, k, mean) ;
```

This function returns a random number with an Erlang statistical distribution. *seed* must be an integer variable initialised with the initial seed value. Each call to the function will update the seed value.

All arguments and return values are integers. In most analog applications the real-valued version `$rdist_erlang` is more appropriate.

#### 4.1.10 `$dist_exponential`

```
integer_value = $rdist_exponential( seed, mean) ;
```

This function returns a random number with an exponential statistical distribution. *seed* must be an integer variable initialised with the initial seed value. Each call to the function will update the seed value.

All arguments and return values are integers. In most analog applications the real-valued version `$rdist_exponential` is more appropriate.

#### 4.1.11 `$dist_normal`

```
integer_value = $rdist_normal( seed, mean, deviation) ;
```

This function returns a random number with an normal statistical distribution. *seed* must be an integer variable initialised with the initial seed value. Each call to the function will update the seed value.

All arguments and return values are integers. In most analog applications the real-valued version `$rdist_normal` is more appropriate.

#### 4.1.12 `$dist_poisson`

```
integer_value = $rdist_poisson( seed, mean) ;
```

This function returns a random number with a poisson statistical distribution. *seed* must be an integer variable initialised with the initial seed value. Each call to the function will update the seed value.

All arguments and return values are integers. In most analog applications the real-valued version `$rdist_poisson` is more appropriate.

#### 4.1.13 `$dist_t`

```
integer_value = $rdist_t( seed, degrees_of_freedom) ;
```

This function returns a random number with a T statistical distribution. *seed* must be an integer variable initialised with the initial seed value. Each call to the function will update the seed value.

All arguments and return values are integers. In most analog applications the real-valued version `$rdist_t` is more appropriate.

#### 4.1.14 `$dist_uniform`

```
integer_value = $rdist_uniform( seed, start, end) ;
```

This function returns a random number with a uniform statistical distribution. *seed* must be an integer variable initialised with the initial seed value. Each call to the function will update the seed value.

All arguments and return values are integers. In most analog applications the real-valued version `$rdist_uniform` is more appropriate.

#### 4.1.15 `$error`

```
$error( [list_of_messages]) ;
```

Does not return a value.

Signals an error condition and terminates the simulation. Unlike `$fatal`, `$error` has no effect if called during a rejected iteration. A message stating the time or step along with the reference of the instance that called the function will be written to the list file and displayed in the command shell.

An optional list of message arguments may also be supplied. These use the same format as the [\\$display](#) system task. The messages are displayed in the command shell and the list file.

If \$error is called in a multi-step simulation performed using multiple cores, each core will act independently. So only the steps for that particular core will terminate.

A call to \$error will set the simulation status to "Error". A subsequent call to the script function GetSimulatorStatus will return 'SimErrors'. (Refer to the *Script Reference Manual* for details.)

## See Also

[\\$fatal](#)

[\\$finish](#)

[\\$info](#)

[\\$warning](#)

### 4.1.16 \$fatal

```
$fatal( finish_code [,list_of_messages] ) ;
```

Does not return a value.

Terminates the simulation immediately. Unlike [\\$error](#), \$fatal will be actioned even if called during an iteration that is rejected.

finish\_code can be 0, 1 or 2 as defined in the following table:

n	Command shell message	List file message
0	none	none
1	Time and instance ref	Time only
2	Time and instance ref	Time and instance ref

An optional list of message arguments may also be supplied. These use the same format as the [\\$display](#) system task. The messages are displayed in the command shell and the list file.

If \$fatal is called in a multi-step simulation performed using multiple cores, each core will act independently. So only the steps for that particular core will terminate.

A call to \$fatal will set the simulation status to "Error". A call to the script function GetSimulatorStatus will return 'SimErrors'.

## See Also

[\\$error](#)

[\\$warning](#)

[\\$info](#)

[\\$finish](#)

### 4.1.17 \$fclose

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

Does not return a value.

Closes one or more file descriptors opened with [\\$fopen](#).

### See Also

[\\$fopen](#)

[\\$fdisplay](#)

[\\$fmonitor](#)

[\\$fdebug](#)

[\\$fwrite](#)

#### 4.1.18 \$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](#)

[\\$fopen](#)

[\\$display](#)

[\\$fdisplay](#)

#### 4.1.19 \$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](#)

[\\$fopen](#)

#### 4.1.20 \$ferror

```
integer_value = $ferror( file_descriptor, returned_message_string ) ;
```

The `$ferror` function may be called after any other function that operates on a file descriptor to return an error code relating to that function. In addition a descriptive string will be written to the `returned_message_string` argument.

The Verilog-A language reference manual does not define the codes that this function should return other than 0 meaning "no error". The string written to `returned_message_string` may be used to display a meaningful message to the user.

if the most recent file operation is a failed call to `$fopen`, the function will return information about the `$fopen` call if zero is passed as the `file_descriptor`.

#### 4.1.21 `$fflush`

```
$fflush( file_descriptor ) ;
```

Does not return a value.

Forces the immediate flushing of a file stream. File writes using `$fwrite` are buffered in memory before being written to disk. This system task forces the buffer to be written out to the file at the end of the current step.

#### 4.1.22 `$fgets`

```
integer count = $fgets(string, file_descriptor) ;
```

Reads a line of text from the file specified by `file_descriptor` and writes it to the string variable `string`. The function returns the number of characters read from the file. If an error occurs the return value is zero. Call `$ferror` to determine the reason for the error.

#### 4.1.23 `$finish`

```
$finish [ ( n ) ] | (n, mode)
```

Does not return a value.

Instructs simulator to finish the analysis. The argument `n` may take the value of 0, 1 or 2 as shown in the table below:

<b>n</b>	<b>Command shell message</b>	<b>List file message</b>
0	none	none
1	Time and instance ref	Time only
2	Time and instance ref	Time and instance ref

If `n` is omitted, it will default to 1.

`$finish` is only effective if called during an iteration that converges and is accepted. Any data generated by the converged iteration will be saved as normal.

The `mode` argument is non-standard and not compatible with other Verilog-A systems. This can have a value of 0 or 1 and determines the behaviour of the function in a multi-step analysis. With a value of 0, or if omitted, the whole simulation will complete and no further steps in the multi-step analysis will be performed. If set to 1, the current step will be terminated but the simulation will continue with the next step.

If `$finish` is called in a multi-step simulation performed using multiple cores, each core will act independently. So, even with a `mode` value of 0, only the steps for that particular core will terminate.

A call to `$finish` is not considered to be an error condition and no error will be flagged. A subsequent call to the script function `GetSimulatorStatus` will return 'Complete'. (Refer to the *Script Reference Manual* for details.) To terminate a simulation and flag an error condition, use either `$error` or `$fatal`.

The LRM states that `$finish` should also trigger the `final_step` event to be executed. SIMetrix does not implement this feature as it is unclear exactly how this should be done. The decision to finish can only be made once the current step has converged as only then are all the values settled to stable values. So it is only at that time that the `final_step` event code can be executed. To do this would require either one more iteration at the current time step or executing a new time step. Neither makes much sense so we decided not to implement this feature.

If compatibility with other Verilog-A systems is required, we recommend not combining `final_step` with the `$finish` system task in the same module.

## See Also

[\\$fatal](#)

[\\$error](#)

[\\$warning](#)

[\\$info](#)

### 4.1.24 \$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*.

### 4.1.25 \$fopen

```
integer file_descriptor = $fopen( filename [, open_mode] ) ;
```

This function returns a file descriptor that can be used for any function that writes to or reads from a file.

If the `open_mode` argument is not supplied, the function operates in multi-channel descriptor mode. Multi-channel descriptors can be used as an argument to functions that write to a file including [\\$fdebug](#), [\\$fdisplay](#), [\\$fmonitor](#), [\\$fstrobe](#) and [\\$fwrite](#).

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 significant) 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.

If the `open_mode` argument is provided and set to "r", the function will open a file in read mode and return an integer that can be passed to functions that can read data from a file including [\\$fscanf](#).

if the `open_mode` argument is provided and set to "w", the function will open a file in write mode and return an integer that can be passed to functions that can write data. This method of creating a write file handle does not suffer from the limit of 31 handles but does not support simultaneous writing of multiple files.

if the `open_mode` argument is provided and set to "a", the behaviour is the same as "w" except that if the file already exists, new output will be appended to the end of the file.

If the same file is opened for write in two or more instances within the same simulation, an independent file handle will be returned that references a temporary file. When the simulation is complete, the temporary files will be merged with the master file.

The file modes "rb", "wb" and "ab" as described in the LRM are also supported but behave identically to "r", "w" and "a" respectively. On read, the ASCII 13 character ("carriage-return") is always ignored and on

write, the CR character is never written out. Currently only text files may be processed using Verilog-A code.

If the specified file cannot be opened, the function returns 0. Note that legitimate file descriptors can be, and usually are, negative. To test whether the file descriptor is valid, be sure to test that it is not equal to zero. Don't test that it is greater than zero.

If the intention is to open a file and keep it open for the duration of an analysis run, the file should be opened in either an `initial_step` event or an analog initial block. It can then be closed in a `final_step` event. Note that it is not possible to close a file then reopen it in the same time step.

The file descriptor should be closed with [\\$fclose](#).

## See Also

[\\$fclose](#)

[\\$fdisplay](#)

[\\$fscanf](#)

### 4.1.26 \$fscanf

```
$fscanf( file_descriptor, format_string, list_of_arguments ) ;
```

Performs the same function as [\\$sscanf](#) but instead of reading characters from a string, reads characters from a file defined by the `file_descriptor` argument. A `file_descriptor` may be obtained using the [\\$fopen](#) function with the "mode" argument set to "r".

### 4.1.27 \$fseek

```
integer result $fseek(file_descriptor, offset, op)
```

Move the file position of the file specified by `file_descriptor` to the position identified by `offset` and `op`. If `op` is 0, `offset` is relative to the start of the file. If `op` is 1, `offset` is relative to the current position and if `op` is 2, `offset` is relative to the end of the file.

The function returns zero if the operation is successful. Otherwise it returns -1. Call [\\$ferror](#) to determine the reason for the error.

Note that if a file is opened in append mode, [\\$fseek](#) is effectively disabled. Append mode does not allow overwriting data already in the file, so the write pointer is always positioned at the end of the file for every write.

## See Also

[\\$ftell](#)

### 4.1.28 \$fstrobe

```
$fstrobe( file_descriptor, format_string, 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](#).

## See Also

[\\$strobe](#)

[\\$display](#)[\\$fopen](#)

### 4.1.29 \$ftell

```
integer file_position $ftell(file_descriptor)
```

Returns the current file position of the file specified by *file\_descriptor*. Returns -1 if an error occurs. Call [\\$ferror](#) to determine the reason for the error.

#### See Also

[\\$ftell](#)

### 4.1.30 \$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](#).

#### See Also

[\\$write](#)[\\$display](#)[\\$fopen](#)

### 4.1.31 \$info

```
$info( [list_of_messages] ) ;
```

Does not return a value.

Writes a message to the list file be considered as a low level warning. [\\$info](#) has no effect if called during a rejected iteration.

The message arguments use the same format as the [\\$display](#) system task.

A call to [\\$info](#) will not change the simulation status. A subsequent call to the script function `GetSimulatorStatus` will return 'Complete'. (Refer to the *Script Reference Manual* for details).

#### See Also

[\\$error](#)[\\$fatal](#)[\\$finish](#)[\\$warning](#)



### 4.1.32 \$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.

### 4.1.33 \$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.

\$monitor writes an implicit new line character at the end of the text.

### See Also

[\\$fmonitor](#)

[\\$display](#)

### 4.1.34 \$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.

### 4.1.35 \$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 BJT with 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).

### 4.1.36 \$random

```
integer_value = $random [ (seed) ] ;
```

Returns a random number. This has three modes of operation according to 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.

Note that this mode of operation is obsolete and may be removed from future releases. To obtain a fixed random value, use `$arandom` assigned to parameter or localparam. Alternatively assign to a variable in an `@initial_step` event or analog initial block.

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.

### 4.1.37 \$rdist\_chi\_square

```
real_value = $rdist_chi_square( [ seed ], degrees_of_freedom[, type_string] ) ;
```

Where *seed* is an integer value and *type\_string* is a string with value "global" or "instance". See `$arandom` for details of the behaviour of *seed* and *type\_string*.

This function returns a random number with a chi square statistical distribution.

### 4.1.38 \$rdist\_erlang

```
real_value = $rdist_erlang( [ seed ], k, mean[, type_string] ) ;
```

Where *seed* is an integer value and *type\_string* is a string with value "global" or "instance". See [\\$arandom](#) for details of the behaviour of *seed* and *type\_string*.

This function returns a random number with an Erlang statistical distribution.

### 4.1.39 \$rdist\_exponential

```
real_value = $rdist_exponential( [ seed ], mean[, type_string] ) ;
```

Where *seed* is an integer value and *type\_string* is a string with value "global" or "instance". See [\\$arandom](#) for details of the behaviour of *seed* and *type\_string*.

This function returns a random number with an exponential statistical distribution.

### 4.1.40 \$rdist\_normal

```
real_value = $rdist_normal( [ seed ], mean, deviation[, type_string] ) ;
```

Where *seed* is an integer value and *type\_string* is a string with value "global" or "instance". See [\\$arandom](#) for details of the behaviour of *seed* and *type\_string*.

This function returns a random number with a normal (or Gaussian) statistical distribution.

### 4.1.41 \$rdist\_poisson

```
real_value = $rdist_poisson( [ seed ], mean[, type_string] ) ;
```

Where *seed* is an integer value and *type\_string* is a string with value "global" or "instance". See [\\$arandom](#) for details of the behaviour of *seed* and *type\_string*.

This function returns a random number with a poisson statistical distribution.

### 4.1.42 \$rdist\_t

```
real_value = $rdist_t( [ seed ], degrees_of_freedom[, type_string] ) ;
```

Where *seed* is an integer value and *type\_string* is a string with value "global" or "instance". See [\\$arandom](#) for details of the behaviour of *seed* and *type\_string*.

This function returns a random number with a T statistical distribution.

### 4.1.43 \$rdist\_uniform

```
real_value = $rdist_uniform( [ seed ], start, end[, type_string] ) ;
```

Where *seed* is an integer value and *type\_string* is a string with value "global" or "instance". See [\\$arandom](#) for details of the behaviour of *seed* and *type\_string*.

This function returns a random number with a uniform statistical distribution.

### 4.1.44 \$rewind

```
$rewind( file_descriptor ) ;
```

\$rewind is exactly the same as [\\$fseek](#)(file\_descriptor, 0, 0).

### 4.1.45 \$sformat

```
$fwrite( string_variable, format_string, list_of_arguments ) ;
```

Does not return a value.

The \$sformat function is similar to \$fwrite with one important difference. \$sformat always interprets its second argument, and only its second argument, as a format string.

The remaining arguments to \$sformat are processed using any format specifiers in the format\_string, until all such format specifiers are used up. If not enough arguments are supplied for the format specifiers or too many are supplied, then a warning will be issued. If the format\_string is a literal string constant, i.e. it is not a variable or parameter, then the mismatch detection will be made at compile-time and in this case an error will be raised.

#### 4.1.46 \$sscanf

```
integer_value = $sscanf(string, format_string, list_of_arguments)
```

\$sscanf decodes an input string according to a defined format to write to one or more variables supplied as arguments. The arguments can be any sequence of strings, integers or reals.

The function will read characters from the format string then match the input string according to the following rules:

1. A white space character in the format string will match zero or more white space characters in the input string
2. A code starting with % in the format string will match a sequence of characters as described in the table below. The % character may followed by an optional sequence of digits representing a maximum width. The sequence of characters up to the maximum width will be read in, decoded then written to the next variable in the argument list
3. Any other non-whitespace character in the format string must match exactly each character in the input string

%d	Integer. Zero or more whitespace characters followed by a sequence of digits with optional '+' or '-' prefix
%e, %f or %g	Real. Floating point real number. Zero or more whitespace characters followed by optional '+' or '-' prefix followed by zero or more digits followed by an optional single decimal point followed by zero or more digits optionally followed by 'e' or 'E' followed by an optional '+' or '-' followed by one or more digits. Floating point value is written to variable
%r	Real. Floating point real number with engineering suffix. Same as %e, %f or %g followed by an optional single engineering suffix, 'a', 'f', 'p', 'n', 'u', 'm', 'k', 'K', 'M', 'G', 'T'. Each suffix scales the value by the conventional factor. (e.g. 'a': 1e-18, 'm': 1e-3, 'M': 1e6)
%s	String. Text string. Zero or more whitespace characters followed by any sequence of characters other than white space
%c	Integer. Single character including white space. ASCII code of character is written to corresponding variable
%m	String. Does not consume any characters from the input string. Writes the hierarchical reference of the instance to the corresponding string variable

Any % codes detected in the format that are not in the above table will be ignored. To match a literal % character, use %%.

\$sscanf will write to variables that are the correct type only. For example, if %s is detected in the format string, the corresponding input string text will only be written to the corresponding variable if the variable is of type string. If the type is incorrect, no assignment will be made. An exception to this is if the data to

be written is an integer and the variable is real. In this instance, the integer value will be written to the real variable.

If there are insufficient variables corresponding to the format string, the input will be consumed but no data will be written. Conversely if there are additional variables with no corresponding codes in the format string, those variables will be ignored.

If the format string is literal (i.e not a variable or model parameter) it will be checked by the compiler at compile time. Any mismatch between the format string and argument list will be detected and reported at that time.

The function will return the number of variables correctly assigned.

#### 4.1.47 \$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 8.00, result will be 0, for 8.30 result will be 30 etc
“simulatorVersion”	Major version of SIMetrix simulator. For version 8.30 this will be 8
“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
“iteration”	Iteration number of the solver. This is the local iteration number and is reset to 1 at the start of each time step or sweep step
“global_iteration”	Global iteration number is incremented on each iteration and reset at the start of the analysis. In a multi-step run it is reset at the start of each step
“startupRamp”	returns value of startup ramp in a transient startup phase. Returns 1.0 normally

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

#### 4.1.48 \$simparam\$str

```
real_value = $simparam$str( string ) ;
```

Similar to [\\$simparam](#) but returns string information. The following strings are supported:

“analysis_type”	Analysis type one of TRAN, DC, AC, SENS, TF, OP
-----------------	---

“analysis_name”	Groupname - e.g. tran1, ac2 etc. Note that the group name is not assigned until after the first analysis point has been evaluated. So this string will <i>not</i> work in an analog initial block or initial_step event.
“cwd”	Returns the directory where the netlist is located
“instance”	Returns the instance name
“netlist_path”	Returns the full path of the netlist
“module”	Returns the module name
“option_name”	Any name that may be used in a .OPTIONS statement and which has a string value

### 4.1.49 \$simprobe

```
real_value = $simprobe( instance_name, variable_name [ , default_value ] ) ;
```

\$simprobe returns the value of an output variable from another instance in the circuit. Output variables (also known as readback variables) are listed in the .out file generated when a simulation is run if ".op" or ".options opinfo" is specified in the netlist. For example, BJTs have an output variable called Gm which reports the gain of the device. The following would return the value of Gm for device Q6:

```
$simprobe( "Q6", "Gm", 0.0 ) ;
```

In the above call, Q6 is assumed to be a "sibling" instance, that is at the same hierarchical level as the calling device. Prefix with '.' to obtain an absolute reference.

If the instance and or parameter cannot be resolved \$simprobe will return the default value supplied. If the default value is omitted, an error will be raised and the simulation will abort.

\$simprobe must be used with care. It should not be called in a manner that closes a loop, for example the result of \$simprobe must not influence the quantity that it is measuring. Convergence failure is likely if this is done. \$simprobe is intended for monitoring and limit testing purposes. So it can be used to report safe operating area violations or it can be used as a probe, but it should not be used as a circuit function.

Including \$simprobe in any Verilog-A module affects the timing of the evaluation of that module. This is because \$simprobe has to wait until the instances that it is monitoring have been fully evaluated. So \$simprobe modules are evaluated later than all others and they are also evaluated in a single non-paralleled thread. This leads to the following recommendations:

\$simprobe should not be used to monitor another instance that itself calls \$simprobe. The returned value may be inaccurate if this is done.

\$simprobe should not be used in large complex modules that might benefit from parallel processing.

### See Also

[OutputVariables](#)

### 4.1.50 \$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](#)

### 4.1.51 \$strobe

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

Does not return a value.

Identical to the [\\$display](#) function.

## See Also

[\\$fstrobe](#)

[\\$display](#)

### 4.1.52 \$swrite

```
$swrite( string_variable, format_string, list_of_arguments ) ;
```

Does not return a value.

Performs the same function as [\\$fwrite](#) except the results is written to a string variable and not to a file.

For details of the format string and how it is interpreted, see the [\\$display](#) function.

### 4.1.53 \$table\_model

```
real_value =
    $table_model( table_inputs, table_data_source, table_control_string) ;
```

`$table_model` implements a multidimensional lookup table. Full details can be found in the language reference manual (LRM) para 9.21 which may be obtained from *Verilog-A Language Reference Manual*. SIMetrix implements the LRM 2.4 specification in full.

## Inputs

The first set of arguments to the function are its inputs, one input for each dimension. So if the table is defined with 2 dimensions, there must be two inputs.

## Data Source

The data to define the table is provided after the inputs. This can either be the name of a file in which case the data is read from that file or it can be entered as a series of arrays. There must be at least N+1 arrays where N is the number of dimensions. The first N arrays define values for the corresponding input while the final array defines the output values. There can be more than N+1 arrays with the arrays actually used being defined by the control string. See below for details.

In the LRM and in the following description, data arrays are referred to as columns.

## Data Source File Format

The data for the table may be defined in a file. The data in the file is arranged in columns with each value separated by a space or tab. Each row of data must occupy a single line. Comments begin with '#' and continue to the end of that line. They may appear anywhere in the file. Blank lines are ignored. The numbers shall be real or integer.

## Control String

The final argument to the function is the control string. This defines a number of options for interpolation, extrapolation and data selection.

The control string is in the form `<code>, <code>, ...; <out-column>`. Each of the `<code>` strings corresponds to an input column. It consists of up to three characters. The first character may be I, D, 1, 2 or 3 and control the interpolation method as defined in the following table:

- I Ignore this input column
- D Discrete lookup. Use the closest point
- 1 Linear interpolation (default)
- 2 Quadratic spline interpolation
- 3 Cubic spline interpolation

The interpolation character may be omitted in which case it defaults to linear interpolation.

The second and third characters define the extrapolation method to use when the input variable overruns the table range. The first character defines the behaviour at the lower end and the second character defines the upper end. The letter may be L, C or E as defined below:

- C Constant extrapolation. The end point value is returned for all input values beyond the table range
- L Linear extrapolation. Linear extrapolation is used with a slope consistent with the interpolation method. See notes for details. This is the default
- E Error. Raise an error and abort if the converged input value exceeds the table range

If only one character is provided, it is applied to both ends of the table. If no extrapolation characters are provided, linear extrapolation is used at both ends.

The `<out-column>` value in the control string defines the column used for the output data. The default is 1 which means the first column encountered after the input data is the output column.

As stated earlier, there must be at least  $N+1$  columns where  $N$  is the number of dimensions. The number of dimensions is determined from the number of input variables supplied. There can be more than one column in which case the 'I' character can be used to mark columns that are ignored for input.

It is legal to provide an empty control string, or miss out the argument altogether, in which case all inputs are defined as "1LL" and the output data is at column  $N+1$  where  $N$  is the number of dimensions.

### Examples

- "1LL,3LL" Assuming 2 dimensions, column 1 defines outer dimension using linear interpolation with linear extrapolation at both ends. Column 2 defines inner dimension using cubic spline interpolation with linear extrapolation at both ends. Output data is in column 3. At least 3 columns must be provided
- "1LL,3LL;2" Assuming 2 dimensions, same as above but output data is read from column 4. Column 3 is unused and at least 4 columns must be provided

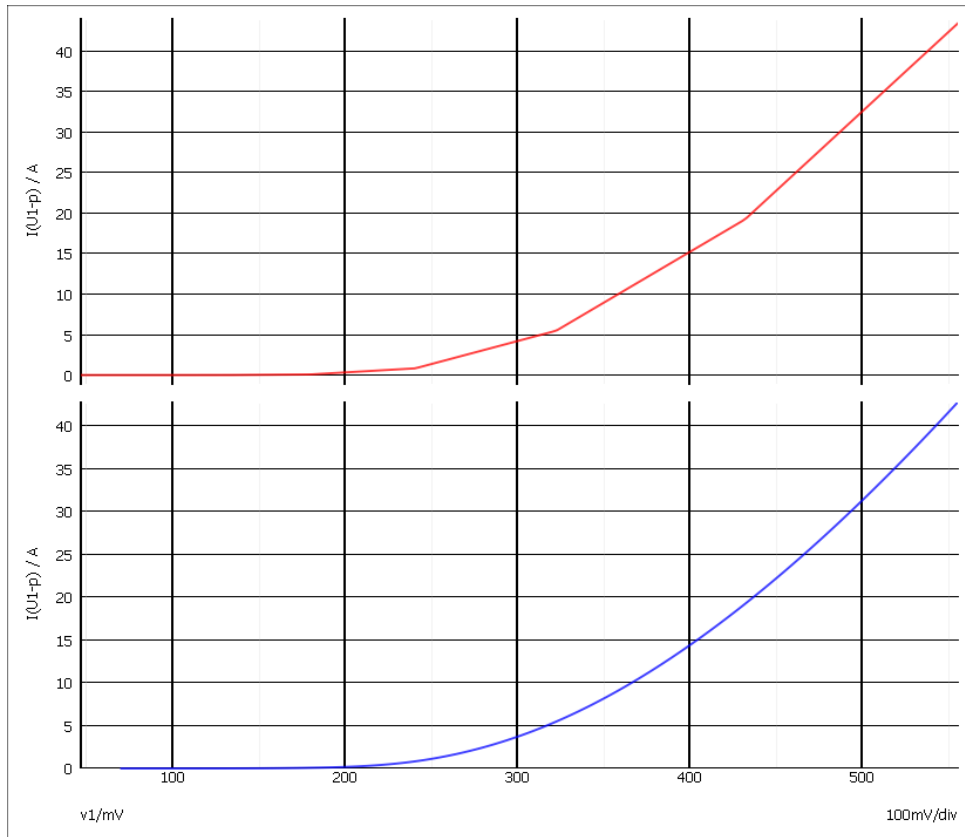


"3LL;2"	Identical to above. First empty code defaults to "1LL"
"EE,3LL;2"	Equivalent to "1EE,3LL;2". As above except outer dimension does not allow excursions outside the data range
"2C,1,3,3"	3 dimensions. Outer dimension uses quadratic spline interpolation with constant extrapolation at both ends. Second dimension uses linear interpolation with linear extrapolation at both ends. Inner dimension uses cubic spline interpolation with linear extrapolation at both ends. Output data is third column after input data, i.e. column 6.
1LC,1LL,I;2	2 dimensions. Outer dimension uses first column in data with linear interpolation, linear extrapolation at the lower end and constant extrapolation at the upper end. Inner dimension is defined in the second column and uses linear interpolation and extrapolation. The third data column is ignored and the output data is taken from the second column after the input data, i.e. column 5. Column 4 is also ignored.

## Interpolation Methods

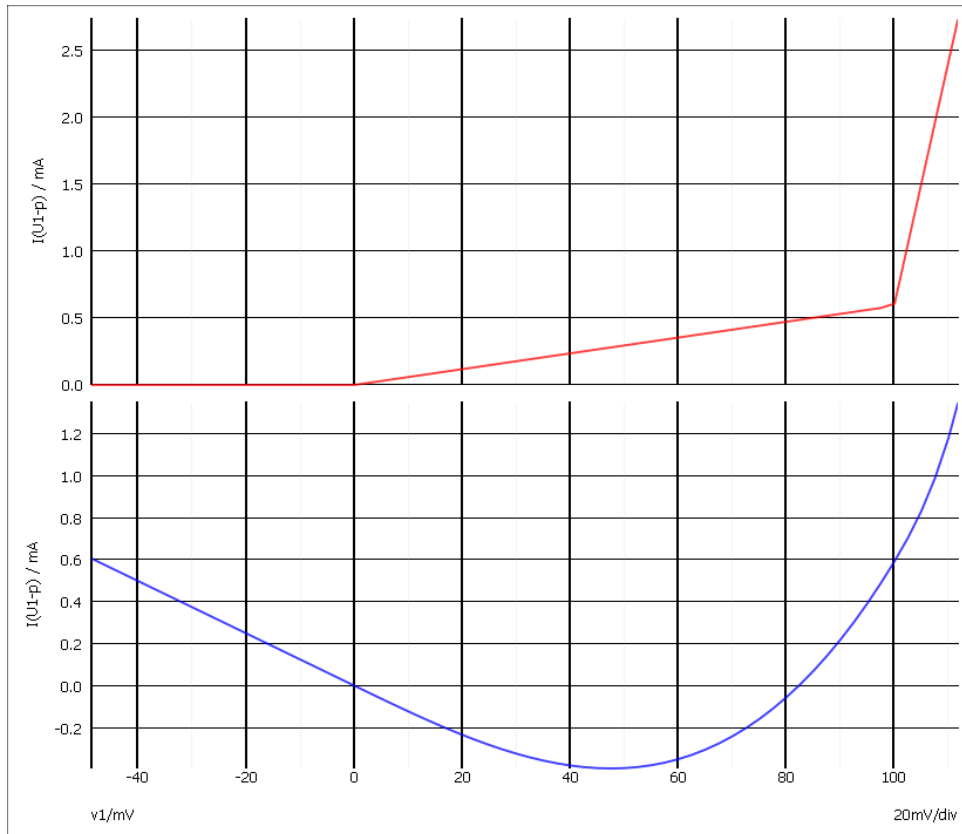
Care must be taken when choosing an interpolation method. While the higher order quadratic and cubic spline methods offer a smoother response, they are both capable of exhibiting instability sometimes with disastrous results. This is especially the case when the input data has abrupt changes in slope. The quadratic spline is considerably more prone to this than the cubic spline. It is recommended that a sample test is performed using a DC sweep to check the behaviour.

To see the dangers of specifying spline interpolation, consider the application of fitting the I-V characteristics of a diode to a \$stable\_model function. Below shows such a device modelled using linear interpolation (top) and cubic interpolation (bottom):



Diode table model - high current

The lower cubic-spline-interpolated curve looks perfectly good and shows a smoother response to the upper linear interpolated response. But now look at the zero crossing:



Diode table model - low current

The upper linear curve faithfully follows the entered data points, but the lower cubic- spline-interpolated curve not only goes negative but its slope also reverse sign. This would give completely erroneous results in a simulation. Providing more closely spaced points around the zero crossing can resolve these problems.

The choice of interpolation method may also impact the simulation performance. For the inner most dimension of the table the burden of calculation is small no matter which interpolation technique is chosen. But the choice can affect how quickly the simulation converges and this is circuit dependent. This can only be assessed by experiment.

However, for higher dimensions in a multi-dimension table, there can be a significant performance penalty using anything other than linear or discrete interpolation. For the higher dimensions, the y-values in the lookup table are derived from interpolating a lower dimension and are thus different on each call. This means that the spline coefficients need to be recalculated on every iteration. For the spline methods, the entire data set needs to be processed to calculate the coefficients even for just a single segment.

## Extrapolation Methods

Three extrapolation options are provided: constant, linear and error. The extrapolation options define how the table function behaves when an input variable exceeds the data range of the table. With constant extrapolation, the end point value is maintained. With linear extrapolation the value follows a straight line with a slope dependent of the interpolation method - see below.

With the error option, the simulation aborts and an error raised if an attempt is made to exceed the input range. Note that the abort only takes place after the iteration has converged and only if the converged result is outside the range. The simulation will not abort if the table input range is exceeded while iterating to convergence but finally converges within range. Linear extrapolation will be selected while iterating to convergence.

The definition of the quadratic and cubic spline interpolators is influenced by the choice of extrapolation. With cubic splines, two constraints, one at each end of the table, need to be set in order to complete the spline definition. If constant extrapolation is selected the constraint is set to be a zero first derivative. If linear extrapolation is set, a zero second derivative is chosen as the constraint and the slope at the end point is chosen as the extrapolation gradient.

For quadratic splines, only a single constraint may be defined. So a zero derivative may be specified at one end but not the other. This leads to a difficulty if both ends are defined with constant extrapolation. The LRM does not define what should be done in this situation. SIMetrix will choose a zero derivative at the lower end, then force continuation of the final point at the upper end. This may lead to a discontinuous slope at the upper end.

#### 4.1.54 \$temperature

```
real_value = $temperature ;
```

Returns the current simulation temperature in Kelvin.

#### 4.1.55 \$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:

$$K = 1.3806226e-23$$

$$q = 1.6021918e-19$$

Currently accepted values:

$$K = 1.380649e-23$$

$$q = 1.602176634e-19$$

#### 4.1.56 \$warning

```
$warning( [list_of_messages] ) ;
```

Does not return a value.

Raises a warning condition. \$warning has no effect if called during a rejected iteration. A message stating the time or step along with the reference of the instance that called the function will be written to the list file.

An optional list of message arguments may also be supplied. These use the same format as the \$display system task. The messages will be written to the list file.

A call to `$warning` will set the simulation status to "Warning". A subsequent call to the script function `GetSimulatorStatus` will return 'Warnings'. (Refer to the *Script Reference Manual* for details). Warnings may be suppressed using the simulator statement `".options nowarnings"`.

## See Also

[\\$fatal](#)

[\\$finish](#)

[\\$info](#)

[\\$error](#)

### 4.1.57 \$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](#)

[\\$display](#)

### 4.1.58 above

```
above( expression [,time_tol [,expr_tol [,enable ]]])
```

`above` is an event function and may only be used in event expressions. It is identical to the [cross](#) event function except that it works in DC analyses and does not have an *edge* argument.

### 4.1.59 abs

```
real_value = abs( x ) ;
```

Returns the absolute value of *x*.

### 4.1.60 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](#).

## See Also

[slew](#)

[transition](#)

### 4.1.61 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.

<i>analysis_name_string</i>	Analysis name in which source is to be active. Currently this must be set to “ac” or be omitted altogether.
mag	<i>Magnitude</i> of source
phase	<i>Phase</i> of source in radians

### 4.1.62 acos

```
real_value = acos( x ) ;
```

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

### 4.1.63 acosh

```
real_value = acosh( x ) ;
```

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

### 4.1.64 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 function 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

#### 4.1.65 asin

```
real_value = asin ( x ) ;
```

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

#### 4.1.66 asinh

```
real_value = asinh( x ) ;
```

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

#### 4.1.67 atan

```
real_value = atan( x ) ;
```

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

#### 4.1.68 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.

#### 4.1.69 atanh

```
real_value = atanh( x ) ;
```

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

#### 4.1.70 ceil

```
real_value = ceil( x ) ;
```

Returns the next integer value greater than  $x$ .

#### 4.1.71 cos

```
real_value = cos( x ) ;
```

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

#### 4.1.72 cosh

```
real_value = cosh( x ) ;
```

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

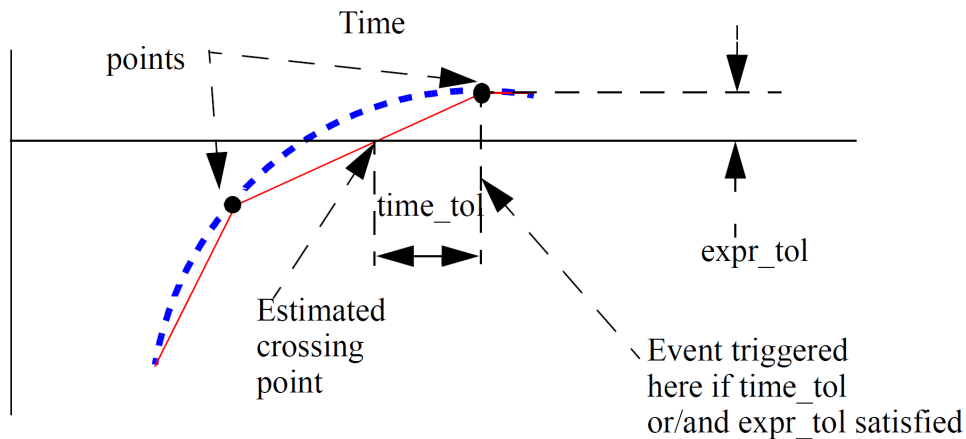
### 4.1.73 cross

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

`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*. 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.
- enable*          if zero, the event function will be disabled

if neither *time\_tol* nor *expr\_tol* are defined or are both set to zero, a default *time\_tol* value will be chosen.



Cross Event Function Behaviour

`cross` stores state information in the same way as an analog operator. It is therefore subject to [Analog Operator Restrictions](#).

### See Also

[timer](#)

[transition](#)



### 4.1.74 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:

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

ddt is an analog operator and is subject to [Analog Operator Restrictions](#).

### See Also

[idt](#)

[idtmod](#)

### 4.1.75 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_identifer( net_or_port_scalar_expression )
```

OR

```
flow_access_identifer( branch_identifer )
```

A *potential\_access\_identifer* is defined in the discipline declarations and is usually 'V' for the electrical discipline. Similarly, the *flow\_access\_identifer* is usually 'I' for the electrical discipline.

*net\_or\_port\_scalar\_expression* can be a module port node or an internal node.

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

### 4.1.76 exp

```
real_value = exp( x ) ;
```

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

### See Also

[limexp](#)

### 4.1.77 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 characteristic 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](#)

### 4.1.78 floor

```
real_value = floor( x ) ;
```

returns the next lower integer to *x*.

#### See Also

[ceil](#)

### 4.1.79 hypot

```
real_value = hypot( x, y ) ;
```

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

### 4.1.80 idt

```
real_value = idt( expression [, initial_condition [, assert [, abstol]] ] ) ;
```

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.

If *assert* is non-zero, the integrator is reset and the return value is the value of *initial\_condition*. Care should be taken when using *assert* as it can result in discontinuous behaviour leading to convergence problems.

*abstol* is accepted but currently is not functional.

`idt` is an analog operator and is subject to [Analog Operator Restrictions](#).

#### See Also

[ddt](#)

[idtmod](#)

### 4.1.81 idtmod

```
real_value = idtmod( expression [, ic [, modulus [, offset [, abstol]]] ] ) ;
```

Returns the time integral of *expression*.

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

If *ic* 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.

*modulus* and *offset*: *modulus* is an expression that must evaluate to a positive value. If not present, idtmod() behaves like [idt\(\)](#). If present, idtmod() returns *k* where  $offset \leq k < modulus$ .

$$\int x(t)dt + ic = n.modulus + k$$

where *n* is an integer and *ic* is the initial condition. *offset* is zero if not provided.

idtmod is an analog operator and is subject to [Analog Operator Restrictions](#).

#### See Also

[ddt](#)

[idt](#)

### 4.1.82 laplace\_nd

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

Where

<i>expr</i>	Input expression.
<i>num_coeffs</i>	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.
<i>den_coeffs</i>	Denominator coefficients in the same format as the numerator - see above.
$\epsilon$	Tolerance parameter currently unused.

The laplace\_nd analog operator implements a Laplace transfer function. This is in the form:

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

where  $d_0, d_1, d_2, \dots, d_m$  are the denominator coefficients and  $n_0, n_1, n_2, \dots, n_m$  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](#).

## See Also

[laplace\\_np](#)

[laplace\\_zd](#)

[laplace\\_zp](#)

### 4.1.83 laplace\_np

```
real_value = laplace_np(expr, num_coeffs, poles [,  $\epsilon$ ] );
```

*expr*            Input expression

*zeros*           Zeros. See [laplace\\_zp](#) for details.

*den\_coeffs*    Denominator coefficients. See [laplace\\_nd](#) for details.

$\epsilon$             Tolerance parameter currently unused

laplace\_np is an analog operator and is subject to [Analog Operator Restrictions](#).

## See Also

[laplace\\_nd](#)

[laplace\\_zd](#)

[laplace\\_zp](#)

### 4.1.84 laplace\_zd

```
real_value = laplace_zd(expr, zeros, den_coeffs [,  $\epsilon$ ] );
```

*expr*            Input expression.

*zeros*           Zeros. See [laplace\\_zp](#) for details.

*den\_coeffs*    Denominator coefficients. See [laplace\\_nd](#) for details.

$\epsilon$             Tolerance parameter currently unused.

laplace\_zd is an analog operator and is subject to [Analog Operator Restrictions](#).

## See Also

[laplace\\_nd](#)

[laplace\\_np](#)

[laplace\\_zp](#)

### 4.1.85 laplace\_zp

```
real_value = laplace_zp(expr, zeros, poles [,  $\epsilon$ ] );
```

*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 denominator 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](#).

## See Also

[laplace\\_nd](#)

[laplace\\_np](#)

[laplace\\_zd](#)

### 4.1.86 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](#).

### 4.1.87 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](#).

## See Also

[exp](#)

### 4.1.88 `ln`

```
real_value = ln( x ) ;
```

Returns the natural logarithm of  $x$ . Range is 0.0 to  $\infty$ .

### 4.1.89 `log`

```
real_value = log( x ) ;
```

Returns the logarithm to base 10 of  $x$ . Range is 0.0 to  $\infty$ .

### 4.1.90 `max`

```
real_value = max( x, y ) ;
```

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

### 4.1.91 `min`

```
real_value = min( x, y ) ;
```

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

### 4.1.92 `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.

### 4.1.93 `sin`

```
real_value = sin( x ) ;
```

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

### 4.1.94 `sinh`

```
real_value = sinh( x ) ;
```

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

### 4.1.95 `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](#).

## See Also

[absdelay](#)

[transition](#)

### 4.1.96 sqrt

```
real_value = sqrt( x ) ;
```

Returns the square root of  $x$ . Range is 0 to  $\infty$ . 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.

### 4.1.97 tan

```
real_value = tan( x ) ;
```

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

### 4.1.98 tanh

```
real_value = tanh( x ) ;
```

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

### 4.1.99 timer

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

`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 than 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 too 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 simulation parameter.

if *enable* is zero, the event function will be disabled.

## See Also

[cross](#)

### 4.1.100 transition

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

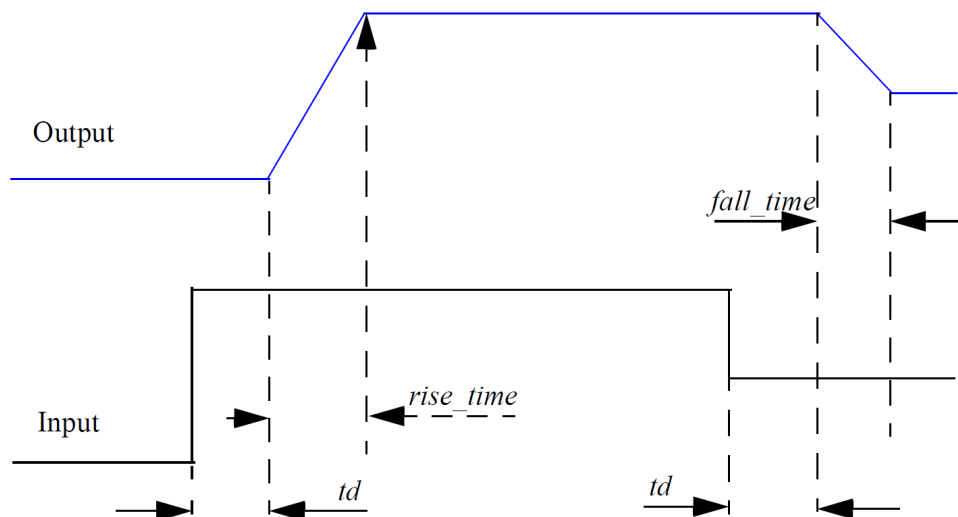
The transition analog operator converts the discrete input value to a continuous output value using specified rise and fall times.

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](#).

## See Also



[absdelay](#)[slew](#)

### 4.1.101 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 characteristic 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](#)

### 4.1.102 zi\_nd

```
real_value = zi_nd(expr, numerator, denominator, interval, [transition, [delay]]) ;
```

The `zi_nd` function implements a linear discrete-time filter defined by z-transform coefficients for both numerator and denominator. Its arguments are:

<i>expr</i>	Input expression
<i>numerator</i>	Array of numerator coefficients in increasing order
<i>denominator</i>	Array of denominator coefficients in increasing order
<i>interval</i>	Sampling interval in seconds
<i>transition</i>	rise and fall time of output at each step
<i>delay</i>	Delay in seconds

The function implements the z-transform:

$$\frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\sum_{k=0}^{N-1} d_k z^{-k}}$$

where  $n_k$  is the  $k^{th}$  numerator coefficient and  $d_k$  is the  $k^{th}$  denominator coefficient.

The numerator and denominator coefficients may be passed to the function either as array parameters or variables or directly as array initialisers. 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. If they are non-constant, the value at the start of the simulation will be used

If *transition* is omitted, a default minimum time is used. If it is set to zero, the transition time will be uncontrolled; in this case the result of this function should be further filtered to avoid convergence issues. The Verilog-A LRM does not allow a z-transform filter to be directly assigned to a branch if a transition time of zero is set. Currently, SIMetrix does not enforce this rule.

## zi\_nd Examples

```
V(out) <+ zi_nd(V(in), {1,-0.9921}, {1,-1.9842,1}, 100u, 1n, 0.0) ;
```

The above implements the z-transform:  $\frac{1 - 0.9921.z^{-1}}{1 - 1.9842.z^{-1} + z^{-2}}$

### 4.1.103 zi\_np

```
real_value = zi_np(expr,numerator,poles,interval,[transition,[delay]]) ;
```

The `zi_np` function implements a linear discrete-time filter defined by z-transform numerator coefficients and by pole locations.

Its arguments are:

<i>expr</i>	Input expression
<i>numerator</i>	Array of numerator coefficients in increasing order
<i>poles</i>	Array of values arranged as real/imaginary pairs defining the transform's pole locations
<i>interval</i>	Sampling interval in seconds
<i>transition</i>	rise and fall time of output at each step
<i>delay</i>	Delay in seconds

The function implements the z-transform:

$$\frac{\sum_{k=0}^{M-1} n_k z^{-k}}{\prod_{k=0}^{N-1} (1 - z^{-1} \rho_k)}$$

where  $n_k$  is the  $k^{th}$  numerator coefficient and  $\rho_k$  is a complex number defining the  $k^{th}$  pole.

The numerator coefficients and pole locations may be passed to the function either as array parameters or variables or directly as array initialisers. 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. If they are non-constant, the value at the start of the simulation will be used.

If *transition* is omitted, a default minimum time is used. If it is set to zero, the transition time will be uncontrolled; in this case the result of this function should be further filtered to avoid convergence issues. The Verilog-A LRM does not allow a z-transform filter to be directly assigned to a branch if a transition time of zero is set. Currently, SIMetrix does not enforce this rule.

## zi\_np Examples

```
V(out) <+ zi_np(V(in), {1,-0.9921}, {0.9921,-0.1253,0.9921,0.1253}, 100u, 1n, 0.0) ;
```

The above implements the z-transform:  $\frac{1 - 0.9921.z^{-1}}{(1 - z^{-1}(0.9921 - 0.1253j))(1 - z^{-1}(0.9921 + 0.1253j))}$

### 4.1.104 zi\_zd

```
real_value = zi_zd(expr,zeros,denominator,interval,[transition,[delay]]) ;
```

The `zi_zd` function implements a linear discrete-time filter defined by z-transform zero locations and by denominator coefficients.

Its arguments are:

<i>expr</i>	Input expression
<i>zeros</i>	Array of values arranged as real/imaginary pairs defining the transform's zero locations
<i>denominator</i>	Array of denominator coefficients in increasing order
<i>interval</i>	Sampling interval in seconds
<i>transition</i>	rise and fall time of output at each step
<i>delay</i>	Delay in seconds

The function implements the z-transform:

$$\frac{\prod_{k=0}^{M-1} 1 - z^{-1} \zeta_k}{\sum_{k=0}^{N-1} d_k z^{-k}}$$

where  $\zeta_k$  is a complex number defining the  $k^{th}$  zero and  $d_k$  is the  $k^{th}$  denominator coefficient.

The zero locations and denominator coefficients may be passed to the function either as array parameters or variables or directly as array initialisers. 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. If they are non-constant, the value at the start of the simulation will be used.

If *transition* is omitted, a default minimum time is used. If it is set to zero, the transition time will be uncontrolled; in this case the result of this function should be further filtered to avoid convergence issues. The Verilog-A LRM does not allow a z-transform filter to be directly assigned to a branch if a transition time of zero is set. Currently, SIMetrix does not enforce this rule.

## zi\_zd Examples

```
V(out) <+ zi_zd(V(in), {0.9921,0}, {1,-1.98422,1}, 100u, 1n, 0.0) ;
```

The above implements the z-transform:  $\frac{1 - 0.9921z^{-1}}{1 - 1.98422z^{-1} + z^{-2}}$

### 4.1.105 zi\_zp

```
real_value = zi_zp(expr,zeros,poles,interval,[transition,[delay]]) ;
```

The *zi\_zp* function implements a linear discrete-time filter defined by z-transform pole and zero locations.

Its arguments are:

<i>expr</i>	Input expression
<i>zeros</i>	Array of values arranged as real/imaginary pairs defining the transform's zero locations
<i>poles</i>	Array of values arranged as real/imaginary pairs defining the transform's pole locations
<i>interval</i>	Sampling interval in seconds
<i>transition</i>	rise and fall time of output at each step
<i>delay</i>	Delay in seconds

The function implements the z-transform:

$$\frac{\prod_{k=0}^{M-1} 1 - z^{-1}\zeta_k}{\prod_{k=0}^{N-1} 1 - z^{-1}\rho_k}$$

where  $\zeta_k$  is a complex number representing the  $k^{\text{th}}$  zero and  $\rho_k$  is a complex number representing the  $k^{\text{th}}$  pole.

## zi\_zp Examples

```
V(out) <+ zi_zp(V(in), {0.9953,0}, {0.9048,0,0.9048,0}, 100u, 1n, 0.0) ;
```

The above implements the following z-transform with a 100us sampling interval and 1ns transition time:

$$\frac{(1 - 0.9953.z^{-1})}{(1 - 0.9048.z^{-1})(1 - 0.9048.z^{-1})}$$

## 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

```
parameter 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](#).

The analog operator restrictions apply to the following functions:

[\\$stable\\_model](#) (But see note in documentation for function)  
[absdelay](#)

cross  
ddt  
idt  
idtmod  
laplace\_nd  
laplace\_np  
laplace\_zd  
laplace\_zp  
last\_crossing  
limexp  
slew  
transition  
zi\_nd  
zi\_np  
zi\_zd  
zi\_zp

# Chapter 5

## Implementation - vs LRM

### 5.1 Overview

Here we describe how SIMetrix Verilog-A compares with the standard as defined in Language Reference Manual 2.4. Full details are below. For SIMetrix extensions, see [SIMetrix Extensions](#).

SIMetrix Verilog-A was originally developed according to the LRM 2.2 standard. In most cases LRM 2.4 is a superset and so a valid SIMetrix Verilog-A module will compile correctly according the LRM 2.4 standard.

### 5.2 SIMetrix Verilog-A vs LRM 2.4

In the following we have highlighted areas where the SIMetrix Verilog-A compiler is not compliant with the LRM 2.4 standard.

#### 5.2.1 2.6 Numbers

All number formats are accepted except sized values, e.g. `4'h1234`. Also the signed prefix (`s` or `S`) is not recognised.

Hex, binary and octal values are allowed. E.g. `'hFF` is 255 as is `'b11111111` and `'o377`. All hex, binary and octal values are unsigned that is the leading bit in such a number is never interpreted as a sign bit. However a unary `'-` may prefix any value.

The LRM allows for the base prefix (e.g. `'b` or `'h`) and the value to be individually macro substituted. This is not currently supported. The number as a whole may be macro substituted but not individual components.

All integer values are represented as signed 32 bit values. This means that the largest unsigned value that can be represented is  $2^{31}-1$ . Any larger value entered will be interpreted as  $2^{31}-1$ ; no error message will be displayed.

#### 5.2.2 2.7 String Literals

String literals are not supported but the string data type is supported. See LRM 2.4, 3.3

### 5.2.3 2.8 Identifiers, Keywords, and System Names

The LRM requires that all identifiers are case sensitive. SIMetrix Verilog-A complies with this except that parameters and instance names cannot be distinguished exclusively by case. For example, this would lead to an error:

```
parameter real BANDWIDTH = 300.0 ;
parameter real bandwidth = 500.0 ;
```

The above is legal Verilog-A but the simulator's parameter handling is not case-sensitive so it would not be possible to distinguish between the above two parameters. However, case-sensitivity is maintained within the Verilog-A module. The following would be allowed:

```
parameter real BANDWIDTH = 300.0 ;
real bandwidth ;
```

Note also that `localparam` objects that differ only by case *are* permitted.

The same rules also apply to instance names used for hierarchical structures.

### 5.2.4 2.9 Attributes

Attributes are supported for variables, parameters and module declarations.

#### 5.2.5 2.9.2 Standard Attributes

`desc`, `units` and `op` attributes applied to a variable declaration are recognised but the function differs slightly from the LRM. `op` is compliant with the LRM and if applied to a variable and set to "yes", the variable will be marked as an output variable and if set to "no" will not be set as an output variable.

For compatibility with older SIMetrix versions, the presence of `desc` or `units` will mark the variable as an output variable as long as `op` is not present.

A output variable is one whose DC operating point value is written to the list file. Data for output variables is also written to the binary data file and may be plotted. See [Output Variables](#)

#### 5.2.6 3.4.2 Parameters - Value Range Specification

Fully compliant. Note that an apostrophe is required before a string value range specification. This changed from LRM 2.2 and some older Verilog-A designs will show a syntax error if the apostrophe is omitted.

#### 5.2.7 3.4.3 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](#).

#### 5.2.8 3.6.2.2 Domain Binding

Anything other than "domain continuous" will raise an error.

### **5.2.9 3.6.2.3 Natureless Disciplines and Domainless Disciplines**

Accepted but non-functional

### **5.2.10 3.6.2.4 Disciplines of Nets and Undeclared Nets**

Not supported

### **5.2.11 3.6.2.7 User Defined Attributes**

Accepted but non-functional

### **5.2.12 3.6.3.1 Net Descriptions**

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

### **5.2.13 3.6.3.2 Net Discipline Initial (Nodeset) Values**

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

### **5.2.14 3.6.5 Implicit Nets**

Not meaningful as hierarchical structures are not yet implemented.

### **5.2.15 3.7 Real Net Declarations**

Not supported in Verilog-A

### **5.2.16 3.8 Default Discipline**

Not supported in Verilog-A

### **5.2.17 3.9 Disciplines of primitives**

Not supported in Verilog-A

### **5.2.18 3.10 Discipline Precedence**

Discipline of all nets is defined in the local module only. Out-of-module references are not implemented.

### **5.2.19 3.11 Net compatibility**

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.



### 5.2.20 3.12 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.

### 5.2.21 4.2.6 Case Equality Operator

Not supported in Verilog-A

### 5.2.22 4.2.13 Concatenations

String concatenations are supported, Numeric concatenation are unsupported.

### 5.2.23 4.2.14 Assignment Patterns

Simple assignment patterns are supported. Replication multiplier is not supported.

### 5.2.24 4.3 Built-in Mathematical Functions

All functions supported. Verilog-HDL style functions with ‘\$’ prefix are now supported.

### 5.2.25 4.5.15 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.

### 5.2.26 4.5.3 Time derivative Operator

Compliant except tolerance is currently ignored.

### 5.2.27 4.5.4 Time integral operator

Fully implemented except *abstol* and *nature* parameters. These parameters are accepted but are non-functional.

### 5.2.28 4.5.5 Circular Integral Operators

Fully implemented except *abstol* and *nature* parameters. These parameters are accepted but are non-functional.

### 5.2.29 4.6.1 Analysis

Compliant except for “nodeset”

### 5.2.30 4.6.2 DC analysis

Compliant except for “nodeset”

### 5.2.31 4.6.4.3 noise\_table

Not implemented

### 5.2.32 4.6.4.4 noise\_table\_log

Not implemented

### 5.2.33 4.7.1 Defining an Analog Function

Compliant except cannot use local parameters

### 5.2.34 4.7.2 Returning a Value from an Analog Function

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

### 5.2.35 5.10.3.4 absdelta Function

Not relevant for Verilog-A

### 5.2.36 6 Hierarchical Structures

Hierarchical structures are currently partially supported. It is possible to instantiate Verilog-A modules provided they satisfy the following:

1. Connections are ordered only. Named connections are not supported
2. Parameters are named only. Ordered parameters are not supported
3. Nodes are scalar. Vectors are not supported
4. `generate` blocks are not used

### 5.2.37 7 Mixed Signal

Not implemented in Verilog-A

### 5.2.38 8 Scheduling Semantics

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

### 5.2.39 9.6 Timescale system tasks

Not relevant for Verilog-A

### 5.2.40 9.7.1 \$finish System Task

\$finish compliant except behaviour of function argument differs from LRM. See [\\$finish](#)

### 5.2.41 9.7.2 \$stop System Task

\$stop compliant except argument to function is ignored. \$stop will cause the simulation to pause in the same way as clicking on the Pause button in the simulator status box. To resume after a call to \$stop, click on the Resume button in the simulator status box or the equivalent menu.

### 5.2.42 9.8

Not relevant for Verilog-A

### 5.2.43 9.9

Not relevant for Verilog-A

### 5.2.44 9.10 Simulator time system functions

\$abstime compliant. \$realtime not supported and now deprecated

### 5.2.45 9.11

Not relevant for Verilog-A

### 5.2.46 9.12

Not relevant for Verilog-A

### 5.2.47 9.13.1 \$random and \$arandom Functions

Compliant except SIMetrix is lenient on usage of type\_string. LRM 2.4 states that the type\_string argument is only accepted for paramset statements used to assign parameter values. SIMetrix allows type\_string to be used anywhere although a warning condition will be raised if used in an analog block.

### 5.2.48 9.13.2 Distribution Functions

All functions implemented. See notes on \$arandom with regard to type\_string parameter.

### 5.2.49 9.15 Analog kernel parameter system functions

\$temperature, \$vt, \$simparam and \$simparam\$str compliant.

**5.2.50 9.17.1 \$discontinuity**

Reports a discontinuity in the main equation if argument is zero. Discontinuities reported in derivatives are ignored. Discontinuities should always be avoided and this function should never be used except as a temporary work around. It will not fix problems caused by discontinuities but will enable some strategies that overcome some of the problems that they cause at the expense of simulation accuracy.

**5.2.51 9.17.3 \$limit**

Compliant using built-in “pnjlim”. User functions not implemented.

**5.2.52 9.18 Hierarchical System Parameter Functions**

\$mfactor implemented. Others are not

**5.2.53 9.19 Explicit binding detection system functions**

Both \$param\_given and \$port\_connected are compliant

**5.2.54 9.20 Analog node alias system functions**

Not implemented

**5.2.55 9.22**

Not relevant for Verilog-A

**5.2.56 9.23**

Not relevant for Verilog-A

**5.2.57 10.1 Compiler directives**

These directives are compliant:

‘define, ‘else, ‘endif, ‘ifdef, ‘include, ‘undef

**5.2.58 10.5 Predefined Macros**

The macro `__VAMS_COMPACT_MODELING__` is always defined indicating that SIMetrix Verilog-A supports the extensions required for supporting compact modelling. This macro is defined in the LRM although the LRM is not clear about what compact modeling extensions are but the support for output variables, the `ddx()` function and extended support for parameters would seem to be important. SIMetrix supports all of these.

The macro `__VAMS_SX_VERSION_GE_840` is always defined and is specific to SIMetrix. It is defined in version 8.4 and later. In general macros of the form `__VAMS_SX_VERSION_GE_majorminor` will always be defined where *major* is the major version number (e.g. 8 for version 8.4) and *minor* is the minor version number (40 for version 8.4). This feature was introduced at version 8.4 and does not apply to versions 8.3 and earlier.

The macro `__VAMS_SX_VERSION` is always defined and has a value equal to *major**minor* where *major* is the major version number (e.g. 8 for version 8.4) and *minor* is the minor version number (40 for version 8.4).

### 5.2.59 11, 12

Not implemented

### 5.2.60 Annex E

SIMetrix Verilog-A supports access to SPICE primitives as described in E.2.2.3. See [SPICE compatibility](#)

## 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 Instance Parameters

The Verilog-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.

### 5.3.3 Boolean Parameters

There is no boolean type in the Verilog-A language and so this data type must be implemented using integer values. SIMetrix and SPICE generally offers a boolean parameter type whereby the value is defined as TRUE if the parameter is present and FALSE if the parameter is not present. The parameter is not assigned a value. It is possible to define a parameter with this type by providing the special attribute *format* and assigning it a value of "flag". For example:

```
(* type="instance", format="flag" *) parameter integer OFF=0 ;
```

In the following instance line, the OFF parameter will have a value of 1:

```
U1 n1 n2 n3 modelname OFF
```

Note that the parameter type must be defined as integer; the attribute will be ignored if any other type.

This feature is not part of the Verilog-A language and will be ignored by other simulators.

### 5.3.4 SPICE compatibility

Verilog-A defines a number of features that allow interfacing between the SPICE simulator and Verilog-A code. SIMetrix implements the ability to instantiate a SPICE device in a Verilog-A module as defined in the LRM 2.4, sections E.2.2.3 and E.3.

## LRM 2.4 Features Supported

SIMetrix Verilog-A supports all primitives as described in Table E.1 of LRM 2.4 with the following exceptions and differences

vsine, isine: parameters supported: dc, mag, phase, offset, ampl, freq, td, sinephase. Other parameters not supported. Additionally supports theta parameter which implements an exponential decay

diode, bjt, mosfet and jfet implement level 1 devices by default. Add a LEVEL parameter to specify other models.

## Additional SPICE Devices

The SIMetrix device internal device name may be used to specify any device type. For a list of available internal names run the built-in script `show_devices` from the SIMetrix command line. This will copy to the clipboard a complete list. The name in the first column may be used provided your license supports it. Note that digital devices may not be used.

### 5.3.5 Output Variables

Output variables are values that can be plotted and are also listed in the .out file if ".op" or ".options opinfo" is specified in the netlist. Further, they can be accessed using the `$simprobe` function.

Normal real-valued variables may be defined as output variables by prefixing the definition with a special attribute. The full form is:

```
(* op="yes" *) real varname ;
```

Optionally, units may also be defined:

```
(* op="yes", units="units" *) real varname ;
```

Where *units* is one of:

```

"V"
"A"
"Secs"
"Hertz",
"Ohm",
"Sie",
"F",
"H",
"J",
"W",
"C",
"Vs",
"V^2",
"V^2/Hz",
"V/rtHz",
"A^2",
"A^2/Hz",
"A/rtHz",
"V/s",
"Celsius"
"coul"
""

```

*description* is an arbitrary description of the variable. Note that *description* is currently unused.

If either description or units is present, the variable will be marked as an output variable.

To plot an output variable, add a .GRAPH statement to the netlist (or F11 window) in the form:

```
.GRAPH instance_ref#variable_name
```

Alternatively to instruct SIMetrix to save the data for the output variable without immediately plotting it, use ".KEEP" instead of ".GRAPH".

### 5.3.6 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 attributes in the following order:

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

Where:

<i>model-type-name</i>	The name used in the .MODEL statement.
<i>level</i>	The LEVEL parameter value in the .MODEL statement.
<i>device-letter</i>	The device letter to use for this device.
<i>default-parameter</i>	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.
<i>version</i>	Value of VERSION parameter.

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

```
(* Mappings="hicum_211,0;nnp,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.

### 5.3.7 Device Binning

Some process development kits (PDK) for IC development use multiple models to define a single transistor type. Different models are selected according to specific instance parameters. This is known as “binning”. Traditionally the selection parameters have been length and width (L) and (W) with model parameters LMIN, LMAX, WMIN and WMAX used to define the range of values allowed for each model bin.

SIMetrix Verilog-A supports model binning. The parameters L, W, LMIN, LMAX, WMIN and WMAX will automatically be used for binning if the parameters are defined in the Verilog-A code.

Other parameters may also be defined using a module attribute. This should follow the following format:

```
(* binparams=binparams_defs *)
```

This should prefix the “module” keyword.

*binparams\_defs* is a semi-colon delimited list of bin parameter definitions. Each bin parameter definition must consist of three parameter names separated by commas. The first is the instance parameter name (e.g. L) and the remaining two parameters the minimum and maximum model parameter names (e.g. LMIN and LMAX).

The traditional , W, LMIN, LMAX, WMIN and WMAX bin parameters can be defined thus:

```
(* binparams="L,LMIN,LMAX;W,WMIN,WMAX" *)
```

Note any bin parameters defined using the above technique will override the built-in L and W definitions. Note also that the actual parameters used must be defined in the Verilog-A code with the bin selection parameter (L and W) defined as an instance parameter and the limit parameters defined as model parameters.

### 5.3.8 Schematic Symbol

When using a Verilog-A module in the schematic editor, a symbol for the module will be needed. The usual method is to invoke the menu **Verilog | Construct Verilog-A Symbol**. This menu auto-generates a symbol based on a simple specification that defines the edge for each pin.

It is possible to decorate the module statement using Verilog-A attributes to control the symbol generation. Two options are available:

- Specify a library symbol instead of an auto-generated symbol. This is useful for modules that define a semi-conductor component such as a diode or transistor. It can also be used to pre-define a specialised symbol



Define the pin positions so as to bypass the GUI that requests this information when the auto-generate menu is invoked

## Specify Library Symbol

To specify a library symbol to be used with a Verilog-A module, add a module attribute with name "symbol" with a value giving a list of symbol names separated by a semicolon:

```
(* symbol="list-of-symbols" *)module module_name(ports) ;
```

For example:

```
(* symbol="npn" *)module bjt_va(c, b, e) ;
```

In the above, the symbol for the NPN transistor will be used to represent the bjt\_va module when the menu **Verilog | Construct Verilog-A Symbol** is invoked.

More than one symbol can be specified in which case the user will be asked which one to use each time the module is placed. For example:

```
(* symbol="npn;pnp" *)module bjt_va(c, b, e) ;
```

will offer the NPN and PNP transistor symbols.

To be able to use a library symbol for a Verilog-A module, the following properties must either be not present on the symbol, or if they are they must be not protected:

MODEL  
MODPARAMS  
MODULENAME  
TEMPLATE  
VAFILE  
VALUE  
VALUESCRIPIT  
VAOPTIONS

About 75% of the symbols in the standard library satisfy this requirement.

The following table lists a number of standard symbols that may be used for various parts. Note this is just a selection; use the Symbol Manager to explore the entire symbol library.

Symbol name	Description	Pins
and2	2-input AND gate	IN1,IN2,OUT
and3	3-input AND gate	IN1,IN2,IN3,OUT
cap_simple_subckt	Capacitor	p,n
dio	Junction diode	p,n
diode_schottky	Schottky diode	p,n
igbt_2	IGBT (no diode)	c,g,e
igbt_with_diode	IGBT (with diode)	c,g,e
ind_simple_subckt	Inductor	p,n
isrc	Fixed current source	p,n

Symbol name	Description	Pins
mov	Varistor	p,n
nand2	2-input NAND gate	IN1,IN2,OUT
nand3	3-input NAND gate	IN1,IN2,IN3,OUT
njfet	N-channel Junction FET	d,g,s
nmf	Gaas FET	d,s,g
nmos	4 terminal NMOS	d,g,s,b
nmos_2gate	2-gate NMOS	s,d,g2,g1
nmos_depletion	4 terminal NMOS depletion type	d,g,s,b
nmos_sub	3 terminal NMOS	d,g,s
nmos_thermal_5	5 terminal thermal NMOS	drain,gate,source,Tj,Tcase
nor2	2-input NOR gate	IN1,IN2,OUT
nor3	3-input NOR gate	IN1,IN2,IN3,OUT
npn	NPN Transistor	c,b,e
npn_darl	NPN darlington	c,b,e
npns	4 terminal NPN (with substrate)	c,b,e,s
opamp	5 terminal opamp	inp,inn,vsp,vsn,out
or2	2-input OR gate	IN1,IN2,OUT
pjfet	P-channel Junction FET	d,g,s
pmos	4 terminal PMOS	d,g,s,b
pmos_thermal_5	5 terminal thermal PMOS	drain,gate,source,Tj,Tcase
pnp	PNP Transistor	c,b,e
pnp_darl	PNP darlington	c,b,e
pnps	4 terminal PNP (with substrate)	c,b,e
res	Resistor (box shape)	p,n
resz	Resistor (z shape)	p,n
scr	Thyristor	A,G,K
switch	Voltage-controlled switch	P,N,CPCN
triac	Triac	MT2,G,MT1
Triode	Thermionic triode with heater	Anode,Grid,Cathode,Heat1,Heat2
ujt	Uni-junction transistor	B2,E,B1
varactor	Varactor (voltage dependent capacitor)	p,n
vsrc	Fixed voltage source	p,n
zener_s	Zener diode	p,n

To use any of the symbols listed above add the attribute as described. The module port names and their order needs to be considered. The symbol will work correctly when assigned by the menu if **either** of the following is satisfied:

The port names match the symbol's pin names exactly

The port names are in the same order as the symbol's pin names (as listed in the above table)

In practice it is best that both are satisfied. However, changing port names for a Verilog-A module that is already written is potentially error-prone and in this case, just making sure they are in the same order as

the symbol's pins will suffice.

## Specify Pin Positions

When an explicit symbol is not specified, the **Verilog | Construct Verilog-A Symbol** menu will auto-generate a symbol. Without any attributes defined a simple dialog will be displayed asking the user to specify on which edge each pin should be located. Rather than imposing the user with this task, it is possible to define pin locations as module attributes.

```
(* pins="list-of-pin-locations" *)module module_name(ports) ;
```

Where list-of-pin-locations is a semicolon-separated list of the letters L,R,T and B representing respectively "Left", "Right", "Top" and "Bottom". There should be one letter for each module port. For example:

```
(* pins="L;L;T;B;R" *)module va_opamp(inp,inn,vcc,vee,out) ;
```

The above will place the inp and inn pins on the left, the vcc pin at the top, the vee pin at the bottom and the out pin on the right.

### 5.3.9 Tolerances

The Verilog-A language only allows for absolute 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, \$schgtol 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.

#### 5.3.10 Analysis() Function

Additional analysis types:

“sens” sensitivity analysis

“tf” transfer function analysis

“pta” pseudo-transient analysis

“smallsig” small signal analysis

“rtn” real time noise analysis

#### 5.3.11 \$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(“reitol”) will return the value of the RELTOL option.

### 5.3.12 \$fopen() Function

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

### 5.3.13 Special Parameters

#### \$model\_dir

The special string parameter \$model\_dir returns the full file system path of the directory containing the file which defines the .MODEL statement associated with a Verilog-A instance. This can be used to locate files that are expected to be in the same folder such as data files for table models. The path returned will use UNIX-style forward slashes as a directory separator and includes a trailing '/’.

Note that this is non-standard and will return an error with other Verilog-A implementations or SIMetrix versions 9.0 or earlier. If using in a Verilog-A module that needs to be compatible with other implementations then use the pre-defined macro \_\_VAMS\_SX\_VERSION\_GE\_910 to selectively compile lines that use that parameter.

## 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 **fully supports** 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.

# Chapter 6

## Debugging with Microsoft Visual Studio

### 6.1 Introduction

In this chapter we describe how to debug the generated 'C' code using Microsoft Visual Studio. The 'C' code follows the Verilog-A code in a sufficiently close manner for this to be a useful debugging technique.

Some knowledge of the 'C' language and the simulator algorithms are required to make full use of this feature but an in-depth understanding is not necessary. It would also be useful if you have some experience with the Microsoft Visual Studio debugger.

Currently Microsoft Visual Studio version 2012, 2013, 2015, 2017, 2019 and 2022, Community, Professional and Enterprise editions are supported. The Build tools editions do not include the graphical debugger and so are not suitable for this application. Also Visual Studio Express available up to and including 2017 is also not supported as it does not properly support 64 bit compilation.

Be aware that the free Community edition is subject to license terms that restrict its use.

Note that we can only offer limited support for the installation and configuration of Visual Studio.

### 6.2 Installing Visual Studio

If you already have a supported version of Visual Studio installed, go to [Checking Compiler](#).

If you do not have Visual Studio installed, you may be able to use the free Community Edition available from Microsoft's website. Note that, at the time of writing (2022) the license terms only permit the use of this product if the annual revenue of your company is less than US\$1m or equivalent in other currencies. Otherwise you will need to acquire a license for Microsoft Visual Studio Professional. In any case you must check the latest license terms at the Microsoft website.

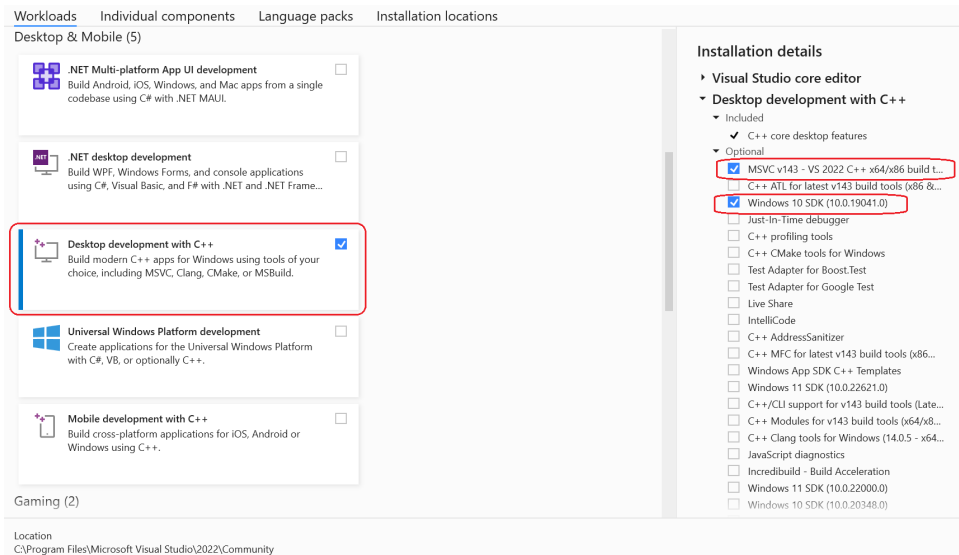
#### 6.2.1 Installing Instruction for Visual Studio 2022

The following explains what is required when installing an edition of Visual Studio 2022.

1. Run the install program as normal
2. Select "Desktop Development with C++" in the "Workloads" page.
3. On the right hand side, expand the item "Desktop Development with C++" then expand the sub-item "Options"
4. Make sure that these two items are checked:

- (a) MSVC v143 - VS2022 C++ x64/x86 build tools (latest)
  - (b) Windows 10 SDK (*version*)
5. Important: if you change the default installation location, SIMetrix will not be able to detect your installation for automatic configuration. It will, however, be possible manually configure the installation.

The picture below shows the minimal installation required for Visual Studio 2022 Community Edition. Enterprise and Professional editions require the same options.

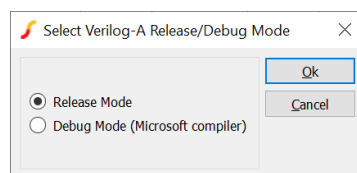


## 6.2.2 Installation Instructions for Visual Studio 2019 and 2017

Visual Studio 2017 and 2019 are installed in a very similar way and use the same install program. The options required are the same but the exact wording and versions are different.

## 6.3 Checking Compiler

Select menu **Verilog | Choose Compiler...**. If your installation of Visual Studio has the necessary components installed and is located at the expected path, you should see the following:



If so all is well and the configuration is complete.

If there is a problem, a message box will show explaining that the installation could not be found. If a supported version and edition of Visual Studio is installed on your system, the error could occur for one of the following reasons:

- C++ command line compilation is not installed. See [Installing Visual Studio](#)
- It is installed to a non-default location. In this case you will need to manually configure the installation. See [Manual Configuration](#)

## 6.4 Manual Configuration

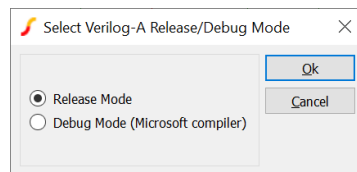
If your installation of Visual Studio is installed to a non-standard path you can manually set up SIMetrix to use your installed Visual Studio. Proceed as follows:

1. Type this command into the SIMetrix command line  
va\_configure\_vs
2. Locate the vcvars64.bat file in your Visual Studio installation. For 2017-2022 it should be located in the subdirectory VC\Auxiliary\Build\. For older versions it should be in subdirectory VC\bin\amd64.

## 6.5 Preparation for Debug

Do the following:

1. Select menu **Verilog | Set Verilog-A Release/Debug Mode....** If you have a supported Microsoft Visual Studio Installation installed you should see the following



select Debug Mode (Microsoft compiler)

If you do not have a supported Microsoft Visual Studio Installation installed or there is a problem with the installation, you will see a message box explaining this. See [Installing Visual Studio](#) above for further details

2. Clear the Verilog-A cache using menu **Verilog | Clear Verilog-A Cache.** This will force you Verilog-A code to be recompiled
3. Run simulation as normal. Instead of the usual messages in the command shell you see with the default compiler, you should see something like this:

```
Microsoft (R) Program Maintenance Utility Version 14.33.31630.0
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
Microsoft (R) C/C++ Optimizing Compiler Version 19.33.31630 for
x64 Copyright (C) Microsoft Corporation. All rights reserved.
```

If the above steps are successful you are ready to start a debug session.

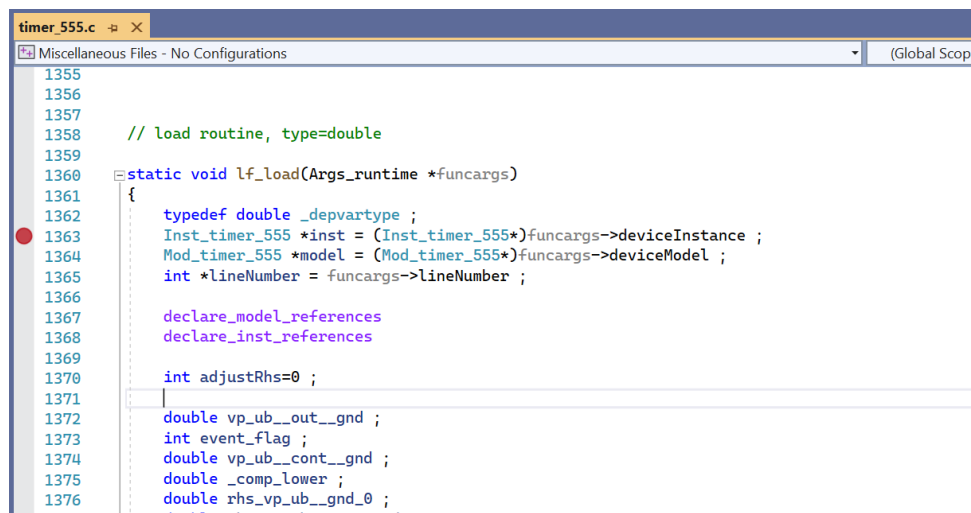
## 6.6 Running a Debug Session

To start a debug session, you first open the 'C' file in Visual studio, then set at least one breakpoint. Next you must attach the Visual Studio debugger to the SIMetrix simulator process which loads the 'C' file.



Finally run the simulation until the breakpoint is hit. Here is the detailed procedure for Visual Studio 2022 and 2019. The procedure for older versions is similar but not identical.

1. Start Visual Studio. The first time this is opened for a particular design, click on Open a local folder
2. Navigate to the folder where the Verilog files are placed. This is  
 C:\Users\[username]\AppData\Roaming\SIMetrix Technologies\SIMetrixnnn\cache\veriloga  
 replace [username] with your login name and replace nnn with the SIMetrix version. E.g. 910 for version 9.1.
3. Open the 'C' file relevant to the module you are debugging. This is always *module\_name.c* where *module\_name* is the name after the **module** statement in the Verilog-A code
4. Set a breakpoint in the `lf_load` function. This is the function that is called to evaluate the device equations on each iteration. Depending on how shortcut keys are setup, this can usually be done with the F9 key. A red filled circle should appear on the left hand side, see picture below:

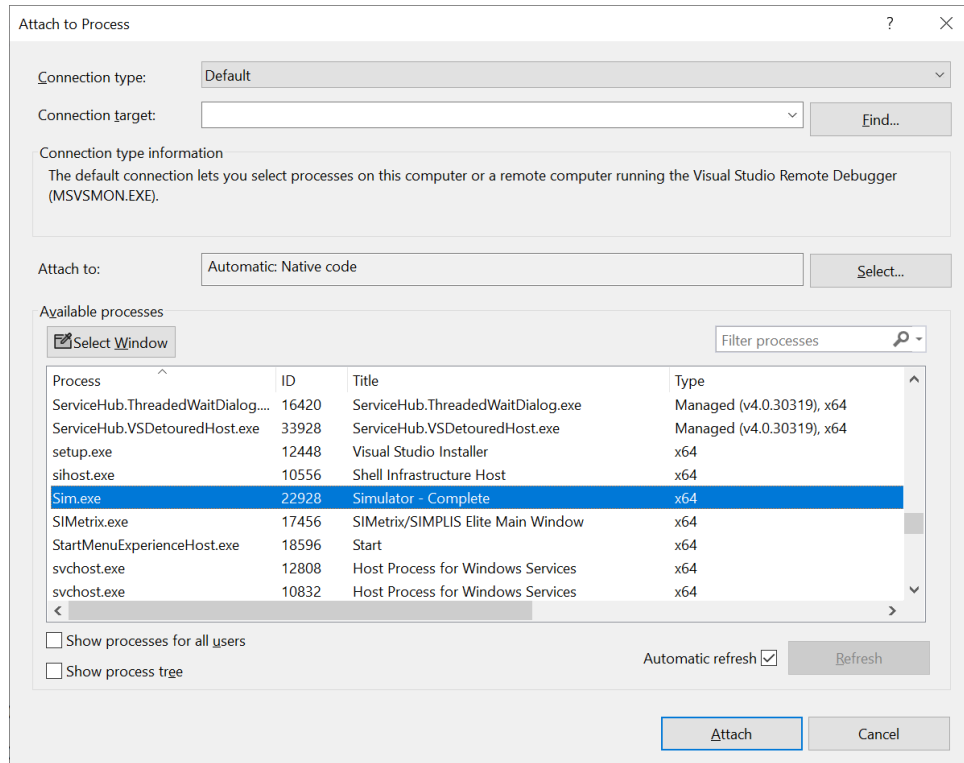


```

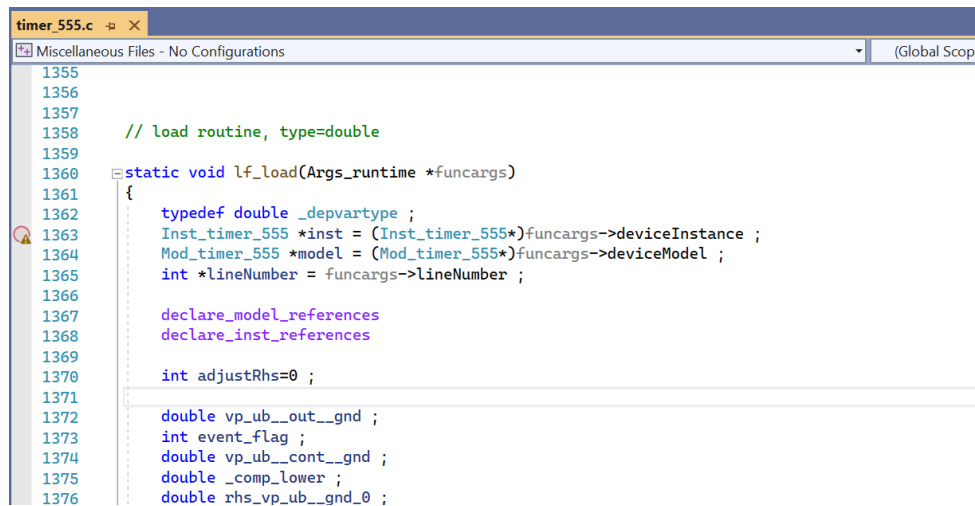
timer_555.c
Miscellaneous Files - No Configurations (Global Scope)
1355
1356
1357
1358 // load routine, type=double
1359
1360 static void lf_load(Args_runtime *funcargs)
1361 {
1362     typedef double _depvartype ;
1363     Inst_timer_555 *inst = (Inst_timer_555*)funcargs->deviceInstance ;
1364     Mod_timer_555 *model = (Mod_timer_555*)funcargs->deviceModel ;
1365     int *lineNumber = funcargs->lineNumber ;
1366
1367     declare_model_references
1368     declare_inst_references
1369
1370     int adjustRhs=0 ;
1371
1372     double vp_ub__out__gnd ;
1373     int event_flag ;
1374     double vp_ub__cont__gnd ;
1375     double _comp_lower ;
1376     double rhs_vp_ub__gnd_0 ;
  
```

5. Select menu **Debug | Attach to Process....**

Navigate to SIM.exe in the Available processes window then click on Attach as shown below:



Note the breakpoint marker after the process is attached. The image might change to a greyed image with an exclamation mark at the corner - see image below:



This means either that the Verilog-A compiled DLL (.sxdev file) is not loaded or it is loaded but was built from a different source file to the one displayed. This may not be a problem if, since running the simulation containing the Verilog-A file being studied, you ran another simulation on another circuit. Running another simulation will unload the .sxdev file from SIM.exe, but when a new simulation on the Verilog-A circuit is started, it will be reloaded. You should see the greyed circle change to a solid red circle when that happens.

## 6. Run simulation

If all is well, the debugger will break at the breakpoint you set. You can now step through the code line by line and look at variable values etc. For more information, see [Debugging Methods](#) below.

## 6.7 Recompiling after Debugging

Once you have run a debug session, you must detach the debugger before recompiling. If you don't do this you will see errors when the Verilog-A code is modified then re-run. This is because the debug symbol file used by the debugger will still be loaded in Visual Studio. Use **Debug | Detach All** to detach the Sim.exe process from the debugger.

Before you reattach Visual Studio, re-run the simulation so that the Verilog-A module is recompiled. It doesn't matter if the simulation fails in some way as long as the Verilog-A compilation completes successfully. Once the compilation is complete you can reattach Sim.exe to Visual Studio and resume your debugging session.

## 6.8 Debugging Methods

The debugger allows you to watch the program flow while examining the values of variables evaluated in the process. However the 'C' code that is executed and debugged is different from the Verilog-A code that you developed. This section explains the structure of the 'C' code and how it relates to the Verilog-A code and provides guidance on how to go about a debugging session.

### 6.8.1 Execution Flow

The compiled Verilog-A code defines how the device equations are evaluated while iterating to a solution. For every time step, the `lf_load` function will execute at least twice (for a linear system) but it is more usual to execute around 4 to 8 times up to a maximum of 40 (controlled by ITL7 option setting).

Once a time step has converged, the time will advance and a new time step will be started. Sometimes the time step is rejected (even if it converged) and the time is reduced and the process repeats.

You can see the current time step as follows:

1. You should see a tabbed window at the bottom labelled Watch 1. Select this to bring into view. If not select menu **Debug | Windows | Watch | Watch 1**

2. In the watch window enter

```
funcargs->time
```

under the name column. You should see the current time show in the value column

Other variables are also available:

Instance name	<code>funcargs-&gt;debug.instName</code>
Model name	<code>funcargs-&gt;debug.modelName</code>
Time step (i.e. the time difference between timepoints)	<code>funcargs-&gt;debug.delta</code>

### 6.8.2 C vs Verilog-A

Let's look at the 'C' code and its corresponding Verilog-A code. Note that the line numbers for the Verilog-A code are reported in the 'C' code using the `tpl_line` macro so we can compare line-by-line.

```

1421     tpl_line(24) ;
1422     is_ub__cont__gnd = 0.0001 * (tpl_mfactor * vp_ub__cont__gnd) ;
1423
1424
1425     tpl_line(25) ;
1426     _ref_low = 0.5 * vp_ub__cont__gnd + vp_ub__gnd_0 ;
1427
1428     tpl_line(28) ;
1429     _reset_thresh = 0.3333333333333333 * vp_ub__vcc__gnd + vp_ub__gnd_0 ;
1430
1431     tpl_line(32) ;
1432     _comp_lower = _ref_low - vp_ub__trig_0 ;
1433
1434     tpl_line(35) ;
1435     event_flag = cross(vp_ub__thresh__cont, 0, 1e-10, 0.0, 1, 0) ;
1436
1437     tpl_line(36) ;
1438     event_flag = cross(_comp_lower, 1, 1e-10, 0.0, 1, 1) ;
1439
1440     tpl_line(39) ;
1441     event_flag = cross(vp_ub__reset__gnd - _reset_thresh, 0, 1e-10, 0.0, 1, 2) ;

```

'C' Code

The Verilog-A code that created the above is:

```

23     I(vcc, cont) <+ v(vcc, cont)/R ;
24     I(cont, gnd) <+ V(cont, gnd)/10k ;
25     ref_low = V(cont, gnd)/2.0+V(gnd) ;
26
27     // Reset threshold - third of VCC
28     reset_thresh = V(vcc,gnd)/3.0+V(gnd) ;
29
30     // Upper and lower comparators.
31     comp_upper = V(thresh, cont) ;
32     comp_lower = ref_low - V(trig) ;
33
34     // "cross" events to catch point where comp thresholds are passed
35     @(cross(comp_upper, 0, 0.1n)) ;
36     @(cross(comp_lower, 1, 0.1n)) ;
37
38     // Ditto reset threshold
39     @(cross(V(reset, gnd)-reset_thresh, 0, 0.1n)) ;
40

```

Verilog-A code

Let's look at some of the 'C' lines and corresponding Verilog-A lines.

#### Line 25

```
ref_low = V(cont, gnd) / 2.0 + V(gnd) ;
```

This translates to 'C' as:

```
_ref_low = 0.5 * vp_ub__cont__gnd + vp_ub__gnd_0 ;
```

You will notice the following:

1. The Verilog-A variable `ref_low` is the same except for a '\_' prefix. In general all user variables are prefixed with an underscore.
2. The probe `v(cont, gnd)` is represented by the 'C' variable `vp_ub__cont__gnd`. This is made up from:

```

vp : voltage probe
ub : unnamed branch
_cont : the Verilog-A cont terminal
_gnd : the Verilog-A gnd terminal

```

Each component is joined by an underscore.

The probe `v(gnd)` is treated in a similar way.

3. The /2.0 is replace \*0.5. In general the Verilog-A compiler combines constants together as a single multiplier where possible

**Line 24**

```
I(cont, gnd) <+ V(cont, gnd)/10k ;
```

Translates to:

```
is_ub__cont__gnd = 0.0001 * (tpl_mfactor * vp_ub__cont__gnd) ;
```

Line 24 is a branch contribution so the variable on the left is a source variable. These are composed in a similar manner to the probe variables described for line 25 above. Instead of the `vp` prefix, it uses the `is` prefix for 'Current source'. Note that probes on the right hand side of an equation or branch contribution do not use the same variable as sources on the left hand side.

The `tpl_mfactor` variable is the value assigned by the 'M' parameter used as a scaling factor. This is almost always 1.0 and you can usually ignore it.

**Line 35**

```
@(cross(comp_upper, 0, 0.1n)) ;
```

Translates to:

```
event_flag = cross(vp_ub__thresh__cont, 0, 1e-10, 0.0, 1, 0) ;
```

Line 35 calls the event function `CROSS`. If the event function triggers, the return value assigned to `event_flag` will be non-zero. In this example the purpose of the event function is to force a time step and no action is taken when the event is triggered. If, however, there was conditional code to be executed at the event, there would be an `if` statement following the event function.

## 6.8.3 Examining Variables

The debugger allows you to see in real-time the values assigned to any variable in the 'C' code. In most cases you can just move the mouse cursor over the variable and its value will be displayed. You can also add them to a "watch list" for a more permanent display.

Note that not all variables in the Verilog-A code will be represented using meaningful names in the 'C' code. If the result of building the full name leads to a name which is too long, a generic name in the form `v_nnn` (nnn is a integer) will be used instead.

## 6.8.4 Partial Derivatives

In many cases, the Verilog-A compiler will not only generate code to evaluate the expression in the Verilog-A file, but it will also generate code to evaluate the partial derivatives of that expression with respect to each of its dependent variables. This is shown in the following code:

```

4443
2124     tpl_line(85) ;
2125     sub127 = _z0 - _k11 ;
2126     sub130 = sub127 * sub127 + _f ;
2127     _z1 = 0.5 * ((_z0 + [k11] - sqrt(sub130)) + _bt ;
2128     dd__z1_vp_ub__node_d__s = 0.5 * (dd__z0_vp_ub__node_d__s - ((dd__z0_vp_ub__node_d__s * 2) * pow(sub127, 1.0)) * tpl_diffsqrt(sub130)) ;
2129
2130     tpl_line(86) ;
2131     sub153 = 1.0 * _g1 * tpl_rmax(vp_ub__node_d__g, 0) ;
2132     _z3 = sqrt(sub153) ;
2133     dd__z3_vp_ub__node_d__g = (_g1 * (tpl_condexpr(vp_ub__node_d__g > 0, 1.0, 0.0))) * tpl_diffsqrt(sub153) ;
2134
2135     tpl_line(87) ;
2136     _z2 = _z1 * _k4 - 1.0 ;
2137     dd__z2_vp_ub__node_d__s = dd__z1_vp_ub__node_d__s * _k4 ;
2138
2139     tpl_line(88) ;
2140     sub162 = _z3 - 1.0 ;
2141     sub167 = dd__z2_vp_ub__node_d__s * _z2 ;
2142     sub171 = 2.0 * (_z2 * _k6) ;
2143     sub176 = 1.0 - 0.5 * (vp_ub__node_d__g * _g1) ;
2144     sub191 = (2.0 * (_k3 - (1.0 + _z2 * _z2) * _k6)) / _g1 ;
2145     _w2 = (sub162 * sub191 + (sub176 * _z3 - 1.0) * sub169) - vp_ub__node_d__g * sub171 ;
2146     dd__w2_vp_ub__node_d__g = (dd__z3_vp_ub__node_d__g * sub191 + (dd__z3_vp_ub__node_d__g * sub176 + (-0.5) * (_g1 * _z3)) * sub169) - sub171 ;
2147     dd__w2_vp_ub__node_d__s = ((-2.0) * (((sub167 + sub167) * _k6) * sub162)) / _g1 - 2.0 * ((dd__z2_vp_ub__node_d__s * _k6) * vp_ub__node_d__g) ;
7110

```

Line 88 for example compiles to 8 lines of 'C'. The first five lines are subexpressions, that is intermediate variables used to evaluate the final result. The final results are the variables `_w2` representing the Verilog-A variable `w2` along with `dd__w2_vp_ub__node_d__g` and `dd__w2_vp_ub__node_d__s`. `dd__w2_vp_ub__node_d__g` and `dd__w2_vp_ub__node_d__s` are partial derivatives of `w2` with respect to  $V(\text{node}_d, g)$  and  $V(\text{node}_d, s)$  respectively. The partial derivatives form part of the iterative equation used to solve the non-linear system.

In most debugging sessions it isn't necessary to track the values of partial derivatives and you can focus on the main equation (`w2` in the above example). However, there are cases where the main equation evaluates successfully but the partial derivatives suffers some numerical issue such as an overflow or divide-by-zero. Such events sometimes lead to a convergence failure but note that this is not always the case. The action of the simulator when an overflow occurs is to reject the time step and try again with a smaller time step. This strategy works more often than it doesn't so a failure to evaluate an equation or its partial derivative does not necessarily require attention.

### 6.8.5 Pre-evaluated Variables

You may notice that some of your Verilog-A code does not appear in the `lf_load` function. The usual reason for this is that the variables were pre-calculated prior to the start of the main simulation run. This happens for variables that are not dependent on any input and assume a constant value throughout the simulation. It is wasteful to re-evaluate such variables for every iteration so these are calculated once at the start.

You will find such variables in either the `lf_temp` or the `lf_modtemp` function. (The name 'temp' here means 'temperature' not 'temporary'. Its use is historic and originates from the original SPICE program where temperature dependent variables were pre-calculated separately to the dynamic variables).

Note that the Verilog-A compiler designates variables that are to be pre-evaluated using a cautious algorithm that may sometimes place the evaluation in `lf_load` unnecessarily.

### 6.8.6 Optimised Variables

There may be some variables that do not appear to be assigned at all. This will be because they have been optimised and typically this happens when a variable has a simple definition, is only used once or is not used at all. The value that was assigned to the variable would have been substituted directly where it was used.

### 6.8.7 Multi-thread Execution

If you have more than one instance of the Verilog-A design in your circuit, the code will be called for each instance. If multi-threading is enabled (which it is by default if the circuit is non-trivial), the execution of the code for the multiple instances will not be sequential. This makes debugging confusing. If this is the case, it is recommended that you disable multi-threading by adding this line to the F11 window:

```
.option mpmnumthreads=1
```

### 6.8.8 Extended and Quad Precision

The Microsoft compiler does not support extended precision, that is the 80bit floating point values used by `.options conv=3` and `.options conv=4`. In 'C' this is usually the data type `long double` but the Microsoft compiler treats `long double` as `double`, i.e. standard 64 bit floating point.

For this reason, Verilog-A modules compiled with the Microsoft compiler will not support extended precision modes and the device code will simply be evaluated to double precision. In most cases this will degrade the precision of the whole circuit not just the Verilog-A modules. For debugging purposes this

will not likely present a problem, but you should remember to switch back to default release mode and recompile after debugging is complete.

The same applies to Quad precision although the reason is a little different. We haven't made the appropriate developments to enable quad precision using the Microsoft compiler; this is not a fundamental limitation of the compiler.

## 6.9 Trademarks

"Microsoft" and "Visual Studio" are trademarks of the Microsoft group of companies.

Copyright © SIMetrix Technologies Ltd. 1992-2024

SIMetrix 9.2 Verilog A Manual