

simetrix

SIMETRIX VERILOG-A MANUAL

VERSION 8.3

APRIL 2019

SIMETRIX VERILOG-A MANUAL

COPYRIGHT © SIMETRIX TECHNOLOGIES LTD. 1992-2019

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

Tel: +44 1635 866395

Fax: +44 1635 868322

Email: support@simetrix.co.uk

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



Contents

1	Introduction	1
1.1	What Is Verilog-A?	1
1.2	Verilog-A Language Reference Manual	1
2	Using Verilog-A Compiler	2
2.1	Using Verilog-A with SIMetrix Schematics	2
2.2	Defining Verilog-A Files in Netlist	2
2.3	Messages	3
2.4	.LOAD Full Syntax	3
2.5	Verilog-A Cache	4
2.6	Permanent .SXDEV Installation	4
3	Writing Verilog-A Code	5
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	Variables	11
3.6.2	\$finish	11
3.6.3	Functions	11
3.6.4	Local Parameters	11
3.6.5	Parameter Limits	12
3.6.6	Conditional Statements	12
3.7	A Capacitor	12
3.8	A Voltage Controlled Oscillator	13
3.9	Digital Elements - Overview	13
3.10	Digital Gate	14
3.10.1	cross() Monitored Event	15
3.10.2	transition() Analog Operator	15
3.11	Butterworth Filter	16
3.11.1	Arrays	18
3.11.2	For Loops	18
3.11.3	laplace_nd Function	18
3.12	RC Ladder	19
3.12.1	Vectors of Nodes	20
3.12.2	Analog For Loops and genvars	20
3.12.3	Compile-time Parameters	20
3.12.4	See Also	21

3.13	Indirect Assignments	21
3.13.1	Virtual Earth	21
3.13.2	Differential Equation	22
4	Verilog-A Reference	24
4.1	Verilog-A Functions	24
4.1.1	\$abstime	26
4.1.2	\$bound_step	26
4.1.3	\$debug	27
4.1.4	\$discontinuity	27
4.1.5	\$display	27
4.1.6	\$fclose	29
4.1.7	\$fdebug	29
4.1.8	\$fdisplay	30
4.1.9	\$fflush	30
4.1.10	\$finish	30
4.1.11	\$fmonitor	30
4.1.12	\$fopen	30
4.1.13	\$fstrobe	31
4.1.14	\$fwrite	31
4.1.15	\$mfactor	31
4.1.16	\$monitor	31
4.1.17	\$param_given	32
4.1.18	\$port_connected	32
4.1.19	\$random	32
4.1.20	\$simparam	33
4.1.21	\$stop	34
4.1.22	\$strobe	34
4.1.23	\$table_model	34
4.1.24	\$temperature	39
4.1.25	\$vt	39
4.1.26	\$write	39
4.1.27	abs	40
4.1.28	absdelay	40
4.1.29	ac_stim	40
4.1.30	acos	41
4.1.31	acosh	41
4.1.32	analysis	41
4.1.33	asin	41
4.1.34	asinh	41
4.1.35	atan	41
4.1.36	atan2	42
4.1.37	atanh	42
4.1.38	ceil	42
4.1.39	cos	42
4.1.40	cosh	42
4.1.41	cross	42
4.1.42	ddt	43
4.1.43	ddx	43
4.1.44	exp	44
4.1.45	flicker_noise	44
4.1.46	floor	44
4.1.47	hypot	45
4.1.48	idt	45
4.1.49	idtmod	45
4.1.50	laplace_nd	46

4.1.51	laplace_np	46
4.1.52	laplace_zd	47
4.1.53	laplace_zp	47
4.1.54	last_crossing	48
4.1.55	limexp	48
4.1.56	ln	48
4.1.57	log	48
4.1.58	max	48
4.1.59	min	49
4.1.60	pow	49
4.1.61	sin	49
4.1.62	sinh	49
4.1.63	slew	49
4.1.64	sqrt	49
4.1.65	tan	49
4.1.66	tanh	50
4.1.67	timer	50
4.1.68	transition	50
4.1.69	white_noise	51
4.1.70	zi_nd	51
4.1.71	zi_np	52
4.1.72	zi_zd	53
4.1.73	zi_zp	54
4.2	Analog Operator Restrictions	55
5	Implementation - vs LRM	57
5.1	Overview	57
5.2	SIMetrix Verilog-A vs LRM 2.4	57
5.2.1	2.7 String Literals	57
5.2.2	2.9 Attributes	57
5.2.3	3.4.2 Parameters - Value Range Specification	57
5.2.4	3.4.3 Parameter Units and Descriptions	57
5.2.5	3.6.2.2 Domain Binding	58
5.2.6	3.6.2.3 Natureless Disciplines and Domainless Disciplines	58
5.2.7	3.6.2.4 Disciplines of Nets and Undeclared Nets	58
5.2.8	3.6.2.7 User Defined Attributes	58
5.2.9	3.6.3.1 Net Descriptions	58
5.2.10	3.6.3.2 Net Discipline Initial (Nodeset) Values	58
5.2.11	3.6.5 Implicit Nets	58
5.2.12	3.7 Real Net Declarations	58
5.2.13	3.8 Default Discipline	58
5.2.14	3.9 Disciplines of primitives	58
5.2.15	3.10 Discipline Precedence	58
5.2.16	3.11 Net compatibility	59
5.2.17	3.12 Branches	59
5.2.18	4.2.6 Case Equality Operator	59
5.2.19	4.2.13 Concatenations	59
5.2.20	4.2.14 Assignment Patterns	59
5.2.21	4.3 Built-in Mathematical Functions	59
5.2.22	4.5.15 Restrictions on Analog Operators	59
5.2.23	4.5.3 Time derivative Operator	59
5.2.24	4.5.4 Time integral operator	60
5.2.25	4.5.5 Circular Integral Operators	60
5.2.26	4.6.1 Analysis	60
5.2.27	4.6.2 DC analysis	60
5.2.28	4.6.4.3 noise_table	60

5.2.29	4.6.4.4 noise_table_log	60
5.2.30	4.7.1 Defining an Analog Function	60
5.2.31	4.7.2 Returning a Value from an Analog Function	60
5.2.32	5.10.3.2 above Function	60
5.2.33	5.10.3.4 absdelta Function	60
5.2.34	6 Hierarchical Structures	61
5.2.35	7 Mixed Signal	61
5.2.36	8 Scheduling Semantics	61
5.2.37	9.6 Timescale system tasks	61
5.2.38	9.7.1 \$finish System Task	61
5.2.39	9.7.2 \$stop System Task	61
5.2.40	9.7.3 \$fatal, \$error, \$warning and \$info System Task	61
5.2.41	9.8	61
5.2.42	9.9	61
5.2.43	9.10 Simulator time system functions	61
5.2.44	9.11	62
5.2.45	9.12	62
5.2.46	9.13.1 \$random Function	62
5.2.47	9.13.2 Distribution Functions	62
5.2.48	9.14 Math System Functions	62
5.2.49	9.15 Analog kernel parameter system functions	62
5.2.50	9.16 Dynamic simulation probe function	62
5.2.51	9.17.1 \$discontinuity	62
5.2.52	9.17.2 \$bound_step	62
5.2.53	9.17.3 \$limit	62
5.2.54	9.18 Hierarchical System Parameter Functions	62
5.2.55	9.19 Explicit binding detection system functions	63
5.2.56	9.20 Analog node alias system functions	63
5.2.57	9.21 Table Based Interpolation and Lookup System Function	63
5.2.58	9.22	63
5.2.59	9.23	63
5.2.60	10 Compiler directives	63
5.2.61	11, 12	63
5.3	SIMetrix Extensions	63
5.3.1	In an Ideal World...	63
5.3.2	Instance Parameters	64
5.3.3	Device Mapping	64
5.3.4	Tolerances	65
5.3.5	Analysis() Function	65
5.3.6	\$simparam() Function	65
5.3.7	\$fopen() Function	65
5.4	Verilog-A Interaction	65
5.4.1	Real-Time Noise	65
5.4.2	Transient Snapshots	66
5.4.3	Pseudo-Transient Analysis	66

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" on page 64](#) for details.

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

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

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

nicenames=0|1 is an advanced feature for debugging purposes. It tells the compiler to use meaningful names in the 'C' file if possible. Otherwise it will 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
Hello World!	Examples/Verilog-A/Manual/Hello-world
A Simple Device Model	Examples/Verilog-A/Manual/Gain-block
A Resistor	Examples/Verilog-A/Manual/Resistor
A Soft Limiter	Examples/Verilog-A/Manual/Soft-limiter
A Capacitor	Examples/Verilog-A/Manual/Capacitor
A Voltage Controlled Oscillator	Examples/Verilog-A/Manual/Vco
Digital Gate	Examples/Verilog-A/Manual/Gates
Butterworth Filter	Examples/Verilog-A/Manual/Butterworth-filter
RC Ladder - Loops, Vectored Nodes and genvars	Examples/Verilog-A/Manual/RC-ladder
Indirect Assignments	Examples/Verilog-A/Manual/Indirect-assignment

3.2 Verilog-A Text Editor

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

3.3 Hello World!

It has become customary to introduce any computer language with a “Hello world” program. That is a program that simply prints “Hello World!”. While Verilog-A was not designed to perform this type of task, it is nevertheless possible to write such a program. Here is an example:

```
module hello_world ;

    analog
    begin

        @(initial_step)
            $strobe("Hello World!") ;

    end
endmodule
```

You can try this using the following procedure:

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

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

If you get any error messages, check the code you entered. The error message should point to the line where the problem occurred. Be aware that sometimes the line number given may not be exact. The point where the parser detects that something is wrong may occur one or two lines after the actual cause of the problem. For example, if you omitted the ‘;’ on the line containing the \$strobe call, you would 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:

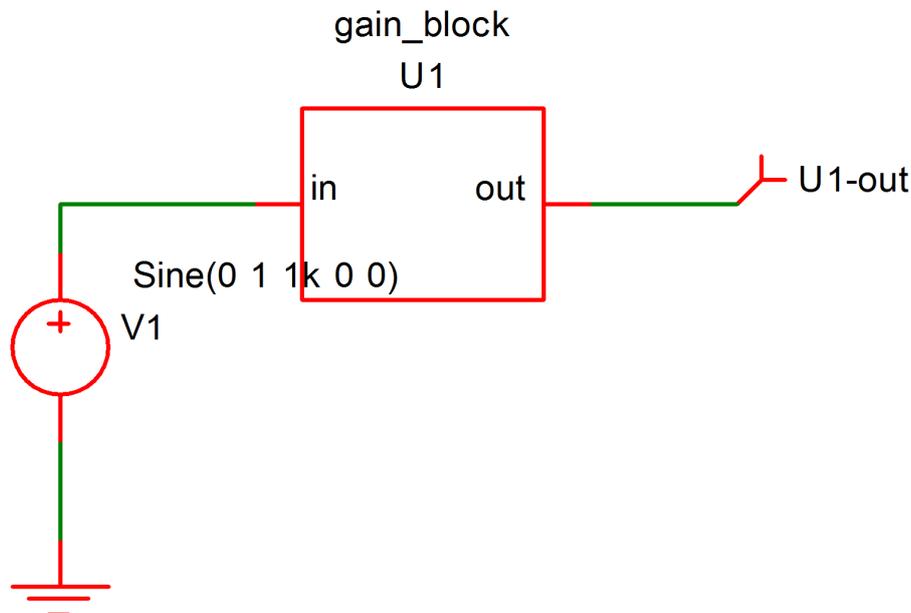
```
`include "disciplines.vams"
module gain_block(in, out) ;

    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 internal node. If two arguments are provided, `V()` accesses the potential between the two nodes. If only a single node is supplied, it accesses the potential between that node and ground.

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

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

3.4.3 Parameters

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

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

```
gain=5
```

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

3.4.4 Disciplines

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

```
electrical in, out ;
```

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

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

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

3.5 A Resistor

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

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

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

Here is the full definition:

```
`include "disciplines.vams"

module va_resistor(p,n) ;

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

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

endmodule
```

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

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

```
`include "disciplines.vams"

module va_resistor(p,n) ;

    parameter real resistance = 1000.0 ;
    electrical p, n ;

    analog
```

```

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

```

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

3.6 A Soft Limiter

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

$$1 - \exp(- (v-vlim))$$

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

```

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

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

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

  real vin ;

  analog
  begin

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

    vin = V(in) ;

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

```

See Examples/Manual/Soft-limiter example

The above example introduces the following new concepts:

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

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

3.6.1 Variables

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

```
real vin ;
```

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

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

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

3.6.2 \$finish

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

3.6.3 Functions

Verilog-A has a range of mathematical functions built-in. In the above example we have used the `exp` function. See "[Verilog-A Functions](#)" on page 24 for a complete list.

3.6.4 Local Parameters

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

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

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

3.6.5 Parameter Limits

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

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

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

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

3.6.6 Conditional Statements

Conditional statements are in the form:

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

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

3.7 A Capacitor

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

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

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

Here is the complete definition for a capacitor:

```
`include "disciplines.vams"

module va_capacitor(p,n) ;

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

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

endmodule
```

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

3.8 A Voltage Controlled Oscillator

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

```

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

module vco(in, out) ;

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

    parameter integer steps_per_cycle=20 ;

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

    electrical in, out ;

    analog
    begin : main

        real vin, instantaneousFreq ;

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

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

    end
endmodule

```

This can be found in Examples/Manual/Vco

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

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

The above can suffer a problem if left to run for a very large number of cycles. The return value from the `idt` operator increases continuously and eventually the size of this value will impact on the calculation precision available leading to inaccuracy. The problem can be resolved using the `idtmod` operator.

3.9 Digital Elements - Overview

Verilog-A can model digital devices as well as analog. This is useful in situations where some simple logic function interfaces mostly with analog elements. An example is a phase detector in a phase-locked loop.

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

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

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

3.10 Digital Gate

Here is a definition for an AND gate

```
`include "disciplines.vams"

module and_gate(in1, in2, out);

    electrical in1, in2, out ;

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

    analog
    begin : main

        integer dig1, dig2, logicState ;

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

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

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

    end
endmodule
```

This example introduces two new concepts:

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

3.10.1 cross() Monitored Event

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

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

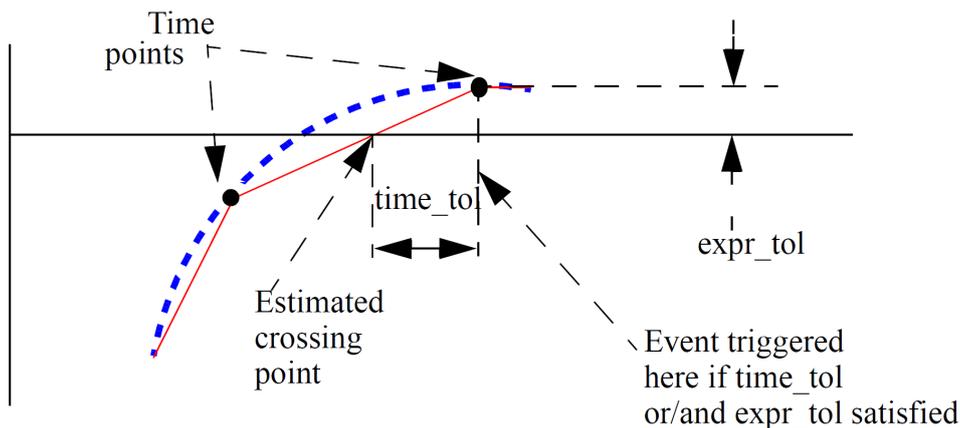
This line both defines the event and also responds to the event when it is triggered. The arguments define the event, while the statement that follows it is the action taken when the event is triggered.

The function has the following form:

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

Only the first argument is compulsory.

<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

3.10.2 transition() Analog Operator

The `transition` function at the end is one of a class of functions called analog operators. The `ddt` and `idt` functions seen earlier are also analog operators. The `transition` analog operator is designed to handle signals that change in discrete steps such as the output of logic devices and digital to analog

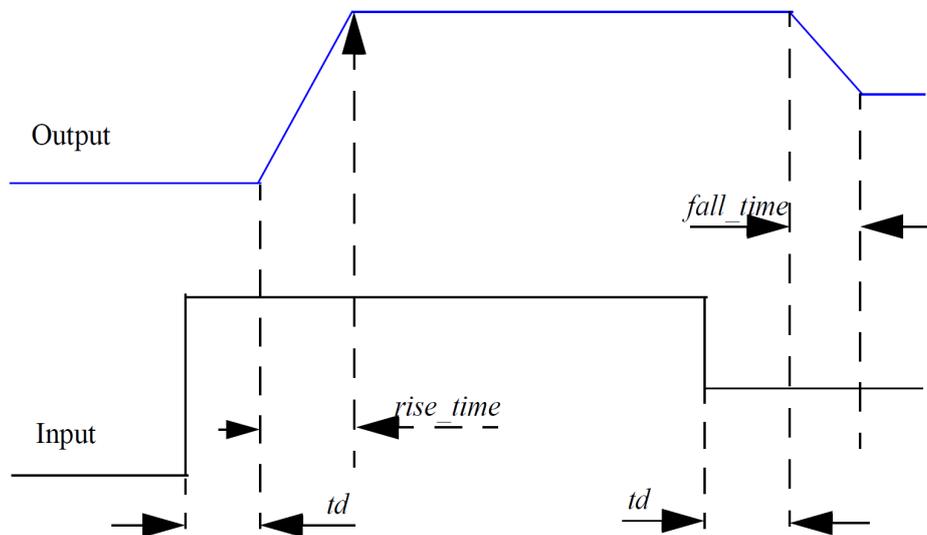
converters. In the and gate example above, the output logic level can change instantaneously but the output of a real device would typically follow a specified rise or fall time. The transition analog operator converts the discrete input value to a continuous output value using specified rise and fall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

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

<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

3.11 Butterworth Filter

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

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

```

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

module laplace_butter(in,ref,out) ;

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

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

    analog
    begin

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

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

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

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

```

See Examples/Manual/Butterworth-filter

This design introduces these language features:

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

3.11.1 Arrays

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

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

Where:

<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 $high_index - low_index + 1$.

3.11.2 For Loops

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

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

<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.

3.11.3 laplace_nd Function

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

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

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_coefs, den_coefs,  $\epsilon$ )
```

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.
ϵ	Tolerance parameter currently unused.

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

3.12 RC Ladder - Loops, Vectored Nodes and genvars

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

```

`include "discipline.h"

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

    electrical [0:n] inode ;

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

    genvar i ;

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

```

This design introduces the following language features:

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

3.12.1 Vectors of Nodes

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

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

1. As a pre-processor constant such as

```
`define n 16
```

this must then be accessed using the back tick character, i.e. ``n`.

2. As a constant `localparam` parameter. These may not be set by the user and so are fixed in value at compile time.
3. As a compile-time parameter. See below for details.

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

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

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

3.12.2 Analog For Loops and `genvars`

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

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

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

SIMetrix will unroll analog for loops if it can. If it can't, because one or more values in the for-loop could not be evaluated at compile-time, it will still attempt to implement the design but this process may fail in which case an error message will be displayed. If it succeeds, a level 2 warning will be raised advising that the design would be more efficient if some variables were constant.

3.12.3 Compile-time Parameters

Compile-time parameters are a SIMetrix extension and not part of the language specification.

Compile-time parameters may be assigned in the `.LOAD` statement in the netlist or they may be defined using an attribute in the Verilog-A code, or both. This concept is in its infancy and we hope to develop it further. The attribute in the code (this is the `(* type="compile_time" *)` prefixing the parameter

keyword) declares the parameter as compile-time and provides a default value. The value may be overridden in the .LOAD statement in the netlist.

3.12.4 See Also

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

3.13 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.

3.13.1 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) ;

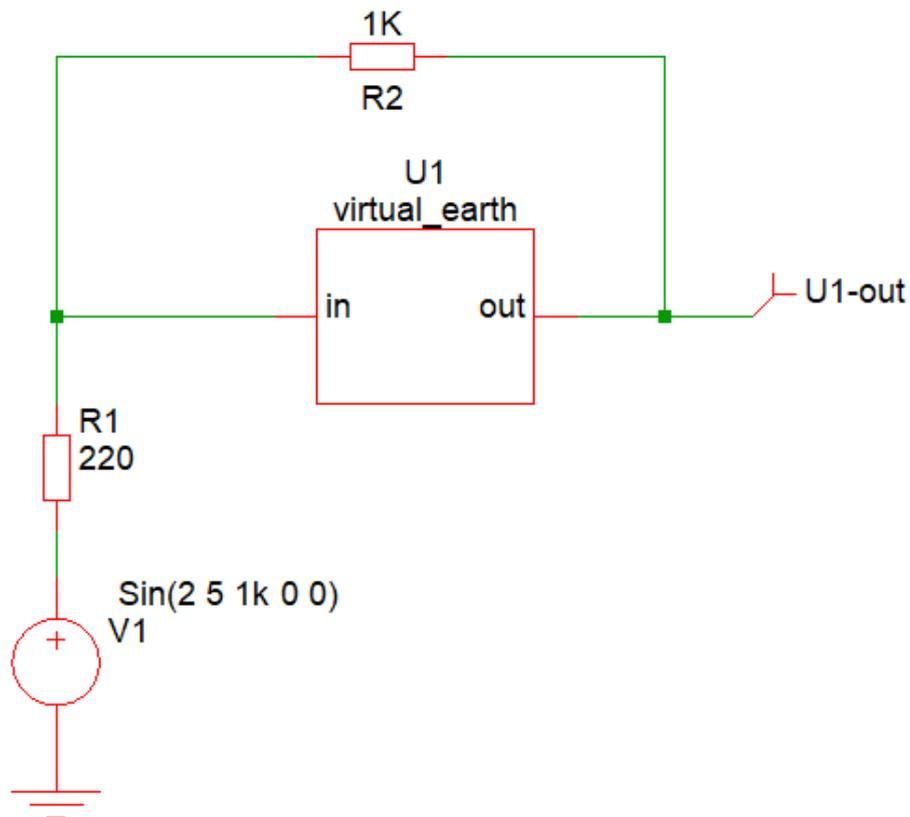
    electrical in, out;

    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 $1k/220$.

3.13.2 Differential Equation

Consider this Verilog-A module:

```
`include "disciplines.vams"

module differential_equation(in, out) ;

    electrical in, out;

    analog
    begin

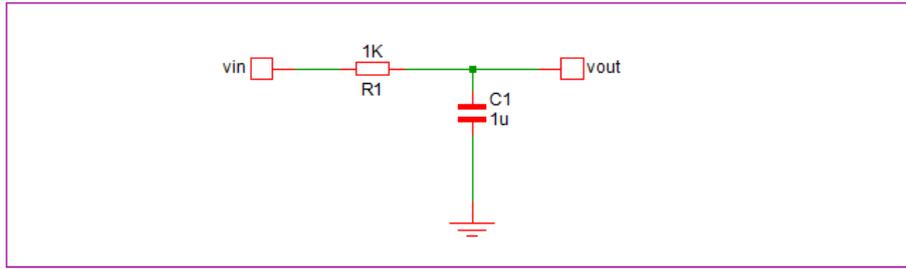
        V(out) : ddt(V(out)) == (V(in) - V(out))/1m ;

    end
endmodule
```

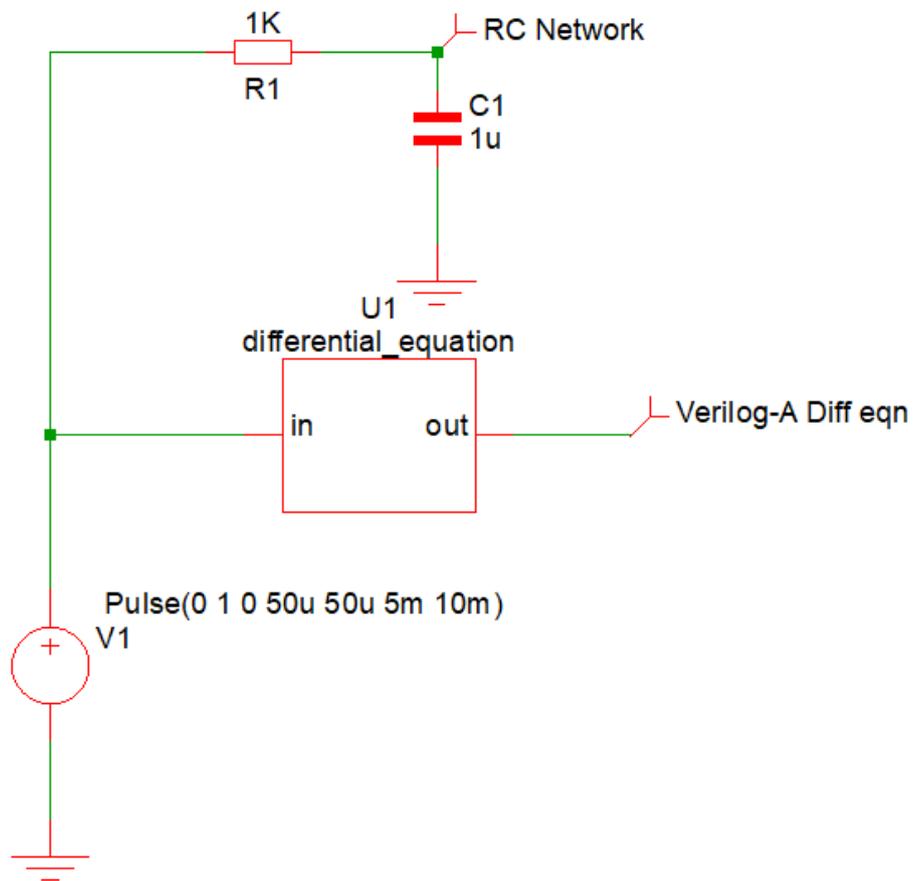
This solves the differential equation:

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

This is the equation that governs the following circuit:



You can try this with this schematic:



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

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

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

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

Does not return a value.

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

4.1.3 \$debug

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

Does not return a value.

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

See Also

[“\\$fdebug” on page 29](#)

[“\\$display” on page 30](#)

4.1.4 \$discontinuity

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

Does not return a value.

Currently \$discontinuity performs no action.

4.1.5 \$display

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

Does not return a value.

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

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

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

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

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

would display:

```
Count=453
```

in the command shell.

The following table shows the format codes available:

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

Real Number Formats

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

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

where:

flag	<p>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</p>
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	<p>e, E, f, F, g, G or r, R</p> <p>e or E: Signed value displayed in exponential format. E.g. 1.23456E3</p> <p>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.</p> <p>g or G: Uses either f or e depending on which is most compact for give precision.</p> <p>r or R: displays in engineering units. Uses these scale factors: T, G, M, K, k, m, u, n, p, f, a.</p>

Notes

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

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

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

See Also

["\\$fdisplay" on page 30](#)

["\\$debug" on page 27](#)

["\\$monitor" on page 31](#)

4.1.6 \$fclose

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

Does not return a value.

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

See Also

["\\$fopen" on page 30](#)

["\\$fdisplay" on page 30](#)

["\\$fmonitor" on page 30](#)

["\\$fdebug" on page 29](#)

["\\$fwrite" on page 31](#)

4.1.7 \$fdebug

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

Does not return a value.

As [\\$debug](#), but writes to a file or files defined by *file_descriptor*.

See Also

["\\$debug" on page 27](#)

["\\$fopen" on page 30](#)

["\\$display" on page 27](#)

["\\$fdisplay" on page 30](#)

4.1.8 \$fdisplay

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

Does not return a value.

As [\\$display](#), but writes to a file or files defined by *file_descriptor*.

See Also

["\\$display" on page 27](#)

["\\$fopen" on page 30](#)

4.1.9 \$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.

4.1.10 \$finish

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

Does not return a value.

Instructs simulator to abort. Currently the argument is ignored.

4.1.11 \$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*.

4.1.12 \$fopen

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

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

There are 31 possible channels each represented by a single bit in the 32 bit returned value. The top (most significant bit) is reserved. The bottom (least 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.

The file descriptor should be closed with `$fclose`.

See Also

[“\\$fclose” on page 29](#)

[“\\$fdisplay” on page 30](#)

4.1.13 \$fstrobe

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

Does not return a value.

As `$strobe`, but writes to a file or files defined by *file_descriptor*. Note that the `$strobe` and `$display` functions are identical. For detailed documentation see [“\\$display” on page 27](#).

See Also

[“\\$strobe” on page 34](#)

[“\\$display” on page 27](#)

[“\\$fopen” on page 30](#)

4.1.14 \$fwrite

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

Does not return a value.

As `$write`, but writes to a file or files defined by *file_descriptor*. Note that the `$write` function is identical to `$display`, except that it does add a new line character. For detailed documentation see [“\\$display” on page 27](#).

See Also

[“\\$write” on page 39](#)

[“\\$display” on page 27](#)

[“\\$fopen” on page 30](#)

4.1.15 \$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.

4.1.16 \$monitor

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

Does not return a value.

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

See Also

[“\\$fmonitor” on page 30](#)

[“\\$display” on page 27](#)

4.1.17 \$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.

4.1.18 \$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).

4.1.19 \$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.

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.

4.1.20 \$simparam

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

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

The \$simparam strings supported by SIMetrix are given as:

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

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

4.1.21 \$stop

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

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

The argument *n* currently has no effect.

See Also

[“\\$finish” on page 30](#)

4.1.22 \$strobe

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

Does not return a value.

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

See Also

[“\\$fstrobe” on page 31](#)

[“\\$display” on page 27](#)

4.1.23 \$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

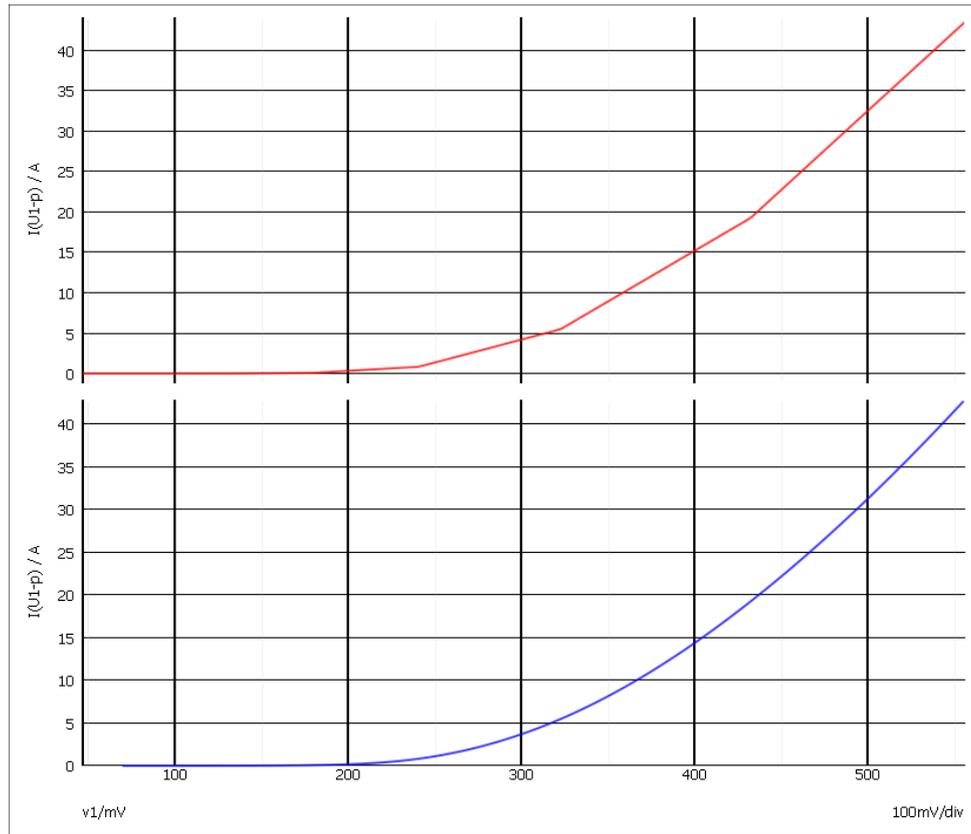
- "1LL,3LL" Assuming 2 dimensions, column 1 defines outer dimension using linear interpolation with linear extrapolation at both ends. Column 2 defines inner dimension using cubic spline interpolation with linear extrapolation at both ends. Output data is in column 3. At least 3 columns must be provided
- "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

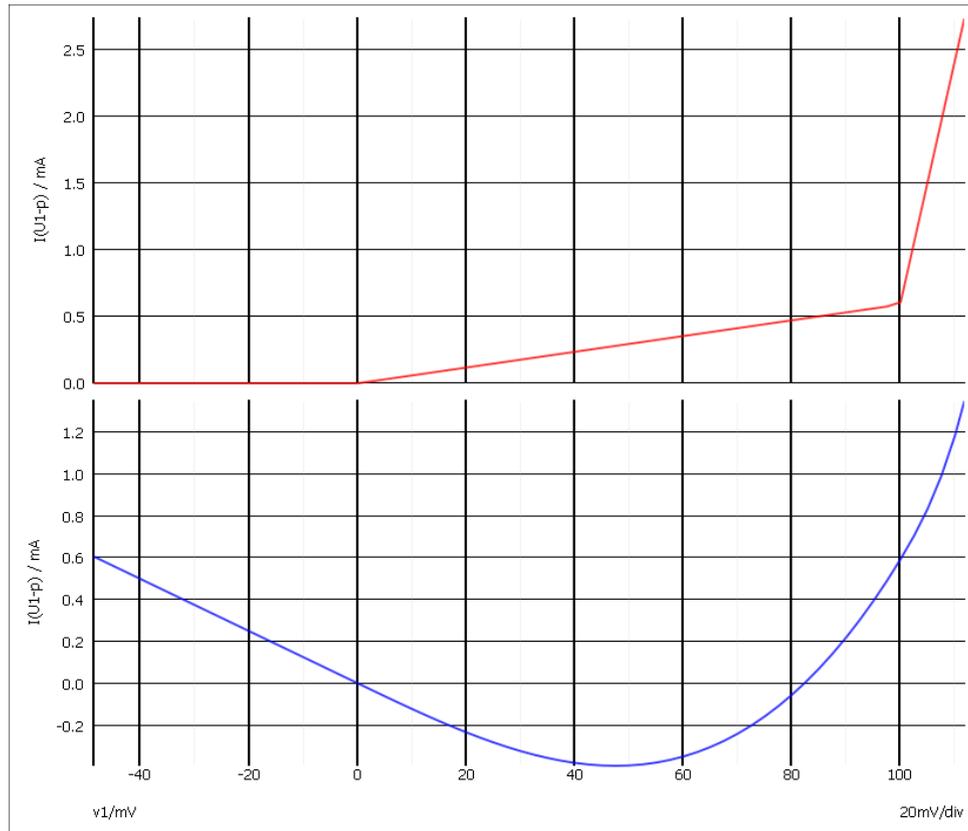
Care must be taken when choosing an interpolation method. While the higher order quadratic and cubic spline methods offer a smoother response, they are both capable of exhibiting instability sometimes with disastrous results. This is especially the case when the input data has abrupt changes in slope. 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.

4.1.24 \$temperature

```
real_value = $temperature ;
```

Returns the current simulation temperature in Kelvin.

4.1.25 \$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:

$$K = 1.3806504e-23$$

$$q = 1.60217646e-19$$

4.1.26 \$write

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

Does not return a value.

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

See Also

["\\$fwrite" on page 31](#)

[“\\$display” on page 27](#)

4.1.27 abs

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

Returns the absolute value of x .

4.1.28 absdelay

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

Applies a transport delay to an input signal.

expression Input signal to delay.

tdelay Delay in seconds. If *maxdelay* is not supplied, only the value of *tdelay* at the start of the simulation will be used and subsequent changes will be ignored. Otherwise changes to *tdelay* will be used as long as they do not exceed *maxdelay*.

maxdelay Maximum delay permitted. If omitted changes to *tdelay* will be ignored. See *tdelay* above.

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

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

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

absdelay is an analog operator and is subject to [“Analog Operator Restrictions” on page 55](#).

See Also

[“slew” on page 49](#)

[“transition” on page 50](#)

4.1.29 ac_stim

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

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

analysis_name_string Analysis name in which source is to be active. Currently this must be set to “ac” or be omitted altogether.

mag Magnitude of source

phase Phase of source in radians

4.1.30 acos

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

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

4.1.31 acosh

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

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

4.1.32 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

4.1.33 asin

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

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

4.1.34 asinh

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

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

4.1.35 atan

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

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

4.1.36 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.

4.1.37 atanh

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

Returns the inverse hyperbolic tangent of x . Range is ± 1.0 .

4.1.38 ceil

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

Returns the next integer value greater than x .

See Also

4.1.39 cos

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

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

4.1.40 cosh

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

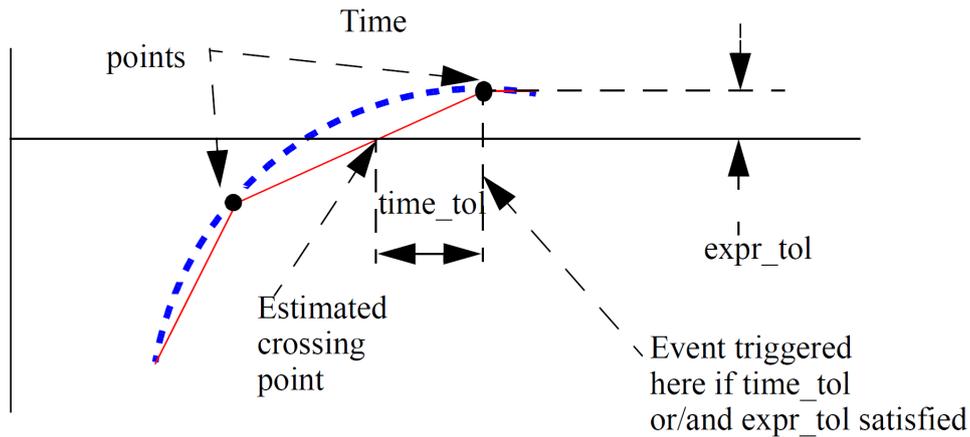
Returns the hyperbolic cosine of x . Range is approx -709 to $+709$.

4.1.41 cross

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

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

<i>expression</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 <code>expr</code> 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 figure below.



Cross Event Function Behaviour

`cross` stores state information in the same way as an analog operator. It is therefore subject to “[Analog Operator Restrictions](#)” on page 55.

See Also

“[timer](#)” on page 50

“[transition](#)” on page 50

4.1.42 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](#)” on page 55.

See Also

“[idt](#)” on page 45

“[idtmod](#)” on page 45

4.1.43 ddx

```
real_value = ddx( expression, unknown_variable) ;
```

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

```
potential_access_identifier( net_or_port_scalar_expression )
```

OR

```
flow_access_identifer( branch_identifier )
```

A *potential_access_identifier* is defined in the discipline declarations and is usually ‘V’ for the electrical discipline. Similarly, the *flow_access_identifer* is usually ‘I’ for the electrical discipline.

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

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

4.1.44 `exp`

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

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

See Also

[“limexp” on page 48](#)

4.1.45 `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” on page 51](#)

4.1.46 `floor`

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

returns the next lower integer to x.

See Also

[“ceil” on page 42](#)

4.1.47 hypot

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

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

4.1.48 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](#)” on page 55.

See Also

[“ddt” on page 43](#)

[“idtmod” on page 45](#)

4.1.49 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\(\)](#)” on page 45. 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](#)” on page 55.

See Also

[“ddt” on page 43](#)

[“idt” on page 45](#)

4.1.50 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.
ϵ	Tolerance parameter currently unused.

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

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

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” on page 55](#).

See Also

[“laplace_np” on page 46](#)

[“laplace_zd” on page 47](#)

[“laplace_zp” on page 47](#)

4.1.51 laplace_np

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

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

laplace_np is an analog operator and is subject to [“Analog Operator Restrictions” on page 55](#).

See Also

[“laplace_nd” on page 46](#)

[“laplace_zd” on page 47](#)

[“laplace_zp” on page 47](#)

4.1.52 laplace_zd

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

- expr* Input expression.
- zeros* Zeros. See [laplace_zp](#) for details.
- den_coeffs* Denominator coefficients. See [laplace_nd](#) for details.
- ε* Tolerance parameter currently unused.

laplace_zd is an analog operator and is subject to [“Analog Operator Restrictions” on page 55](#).

See Also

[“laplace_nd” on page 46](#)

[“laplace_np” on page 46](#)

[“laplace_zp” on page 47](#)

4.1.53 laplace_zp

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

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

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

where *re* is the real part and *im* imaginary part. If a zero is complex (i.e. the imaginary part is non-zero) then its complex conjugate must also be present. If both real and imaginary parts are zero then the zero becomes just *s*.

The values can be entered as an array variable or as an array initialiser. An array initialiser is a sequence of comma separated values enclosed with ‘{’ and ‘}’. E.g: 1.0, 2.3, 3.4, 4.5. The values do not need to be constants.

- poles* Array of pairs of real numbers representing the poles of the Laplace transform. Each pair consists of a real part and an imaginary part with the real part first. Each pole introduces a product term on the 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” on page 55](#).

See Also

[“laplace_nd” on page 46](#)

[“laplace_np” on page 46](#)

[“laplace_zd” on page 47](#)

4.1.54 last_crossing

```
real_value = last_crossing( expression, direction ) ;
```

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

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

last_crossing is an analog operator and is subject to [“Analog Operator Restrictions” on page 55](#).

4.1.55 limexp

```
real_value = limexp( x ) ;
```

Returns the exponential of *x*. limexp limits its change in output from iteration to iteration in order to improve convergence. In situations where its return value is not the true exponential of *x* it will force further iterations. The iteration will only be accepted when the result is the true value of $\exp(x)$. Thus, limexp can be seen as a direct replacement for exp but with improved convergence. But note that limexp is an analog operator and is therefore subject to [“Analog Operator Restrictions” on page 55](#).

See Also

[“exp” on page 44](#)

4.1.56 ln

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

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

4.1.57 log

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

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

4.1.58 max

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

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

4.1.59 min

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

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

4.1.60 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.

4.1.61 sin

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

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

4.1.62 sinh

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

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

4.1.63 slew

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

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

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

slew is an analog operator and is subject to “Analog Operator Restrictions” on page 55.

See Also

[“absdelay” on page 40](#)

[“transition” on page 50](#)

4.1.64 sqrt

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

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

4.1.65 tan

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

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

4.1.66 tanh

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

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

4.1.67 timer

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

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

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

statement is executed when the event is triggered.

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

The event is first scheduled at *time*. If *period* is specified and is greater 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.

See Also

[“cross” on page 42](#)

4.1.68 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.

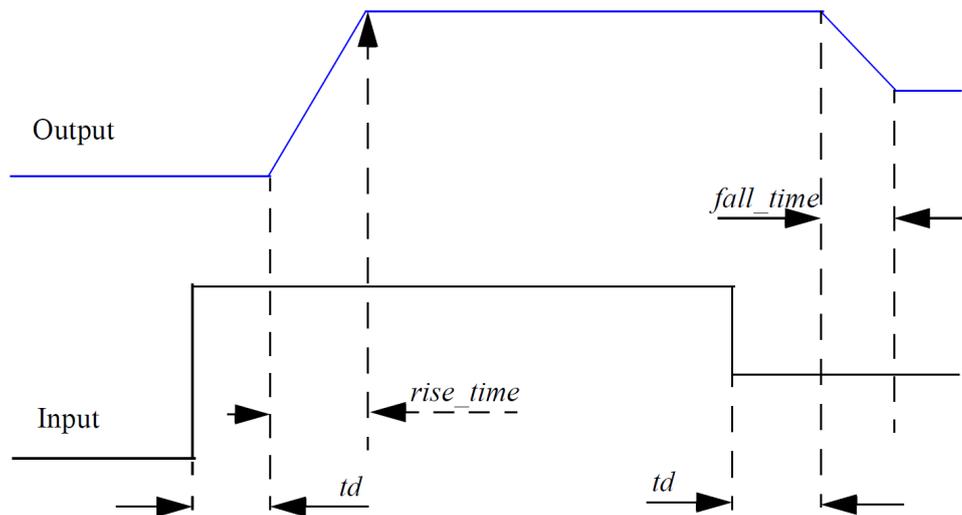
Its arguments are:

<i>expr</i>	Input expression.
<i>td</i>	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.
<i>rise_time</i>	Rise time of output in response to change in input.
<i>fall_time</i>	Fall time of output in response to change in input.
<i>time_tol</i>	Currently ignored.

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

Manual/Command Reference/.OPTIONS for details of MINBREAK.

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



transition is an analog operator and is subject to [“Analog Operator Restrictions”](#) on page 55.

See Also

[“absdelay”](#) on page 40

[“slew”](#) on page 49

4.1.69 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”](#) on page 44

4.1.70 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: $\frac{1 - 0.9921.z^{-1}}{1 - 1.9842.z^{-1} + z^{-2}}$

4.1.71 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 ρ_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: $\frac{1 - 0.9921 \cdot z^{-1}}{(1 - z^{-1}(0.9921 - 0.1253j))(1 - z^{-1}(0.9921 + 0.1253j))}$

4.1.72 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 ζ_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: $\frac{1 - 0.9921.z^{-1}}{1 - 1.98422z^{-1} + z^{-2}}$

4.1.73 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 ζ_k is a complex number representing the k^{th} zero and ρ_k is a complex number representing the k^{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](#)” on page 20.

The analog operator restrictions apply to the following functions:

“[\\$table_model](#)” on page 34 (But see note in documentation for function)

“[absdelay](#)” on page 40

“[cross](#)” on page 42

“[ddt](#)” on page 43

“[idt](#)” on page 45

“[idtmod](#)” on page 45

“[laplace_nd](#)” on page 46

“[laplace_np](#)” on page 46

“[laplace_zd](#)” on page 47

“[laplace_zp](#)” on page 47

“[last_crossing](#)” on page 48

“[limexp](#)” on page 48

“[slew](#)” on page 49

“[transition](#)” on page 50

[“zi_nd” on page 51](#)

[“zi_np” on page 52](#)

[“zi_zd” on page 53](#)

[“zi_zp” on page 54](#)

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” on page 63](#).

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 2.7 String Literals

String variables are not supported.

5.2.2 2.9 Attributes

Attributes are supported for variables, parameters and module declarations.

5.2.3 3.4.2 Parameters - Value Range Specification

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

5.2.4 3.4.3 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” on page 64](#).

5.2.5 3.6.2.2 Domain Binding

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

5.2.6 3.6.2.3 Natureless Disciplines and Domainless Disciplines

Implemented but non-functional

5.2.7 3.6.2.4 Disciplines of Nets and Undeclared Nets

Not supported

5.2.8 3.6.2.7 User Defined Attributes

Accepted but non-functional

5.2.9 3.6.3.1 Net Descriptions

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

5.2.10 3.6.3.2 Net Discipline Initial (Nodeset) Values

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

5.2.11 3.6.5 Implicit Nets

Not meaningful as hierarchical structures are not yet implemented.

5.2.12 3.7 Real Net Declarations

Not supported in Verilog-A

5.2.13 3.8 Default Discipline

Not supported in Verilog-A

5.2.14 3.9 Disciplines of primitives

Not supported in Verilog-A

5.2.15 3.10 Discipline Precedence

Not meaningful as hierarchical structures are not yet implemented.

5.2.16 3.11 Net compatibility

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

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

5.2.17 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.

5.2.18 4.2.6 Case Equality Operator

Not supported in Verilog-A

5.2.19 4.2.13 Concatenations

Not supported

5.2.20 4.2.14 Assignment Patterns

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

5.2.21 4.3 Built-in Mathematical Functions

Currently the Verilog-HDL versions of standard functions which start with a '\$' are not supported.

5.2.22 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.

5.2.23 4.5.3 Time derivative Operator

Compliant except tolerance is currently ignored.

5.2.24 4.5.4 Time integral operator

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

5.2.25 4.5.5 Circular Integral Operators

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

5.2.26 4.6.1 Analysis

Compliant except for “nodeset”

5.2.27 4.6.2 DC analysis

Compliant except for “nodeset”

5.2.28 4.6.4.3 noise_table

Not implemented

5.2.29 4.6.4.4 noise_table_log

Not implemented

5.2.30 4.7.1 Defining an Analog Function

Compliant except cannot use local parameters

5.2.31 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.

5.2.32 5.10.3.2 above Function

Not implemented

5.2.33 5.10.3.4 absdelta Function

Not relevant for Verilog-A

5.2.34 6 Hierarchical Structures

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

This section of the standard does include the syntax for module definitions and this is of course fully supported. This is covered by the opening paragraphs of section 6.2. The rest of the section is not implemented.

5.2.35 7 Mixed Signal

Not implemented in Verilog-A

5.2.36 8 Scheduling Semantics

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

5.2.37 9.6 Timescale system tasks

Not relevant for Verilog-A

5.2.38 9.7.1 \$finish System Task

\$finish compliant except argument to function is ignored

5.2.39 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.

5.2.40 9.7.3 \$fatal, \$error, \$warning and \$info System Task

Not implemented

5.2.41 9.8

Not relevant for Verilog-A

5.2.42 9.9

Not relevant for Verilog-A

5.2.43 9.10 Simulator time system functions

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

5.2.44 9.11

Not relevant for Verilog-A

5.2.45 9.12

Not relevant for Verilog-A

5.2.46 9.13.1 \$random Function

Currently supports the LRM 2.2 specification with the first argument only. Version 2.4 LRM defines two functions, \$random and \$arandom. Refer to “[\\$random function](#)” on page 32 for details about the SIMetrix implementation of the \$random function.

5.2.47 9.13.2 Distribution Functions

Not supported

5.2.48 9.14 Math System Functions

Verilog-HDL versions of regular Verilog-A math functions. These are not currently supported but most functions can be used by simply removing the \$ prefix.

5.2.49 9.15 Analog kernel parameter system functions

\$temperature, \$vt, \$simparam compliant. \$simparam\$str is not supported

5.2.50 9.16 Dynamic simulation probe function

Not implemented

5.2.51 9.17.1 \$discontinuity

Accepted but non-functional

5.2.52 9.17.2 \$bound_step

Compliant

5.2.53 9.17.3 \$limit

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

5.2.54 9.18 Hierarchical System Parameter Functions

\$mfactor implemented. Others are not

5.2.55 9.19 Explicit binding detection system functions

Both \$param_given and \$port_connected are compliant

5.2.56 9.20 Analog node alias system functions

Not implemented

5.2.57 9.21 Table Based Interpolation and Lookup System Function

Implementation is based on LRM 2.2 definition which supports linear interpolation only.

5.2.58 9.22

Not relevant for Verilog-A

5.2.59 9.23

Not relevant for Verilog-A

5.2.60 10 Compiler directives

These directives are compliant:

'define, 'else, 'endif, 'ifdef, 'include, 'undef

Predefined macros are not implemented

5.2.61 11, 12

Not implemented

5.3 SIMetrix Extensions

5.3.1 In an Ideal World...

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

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

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

5.3.2 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 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.

5.3.4 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.

5.3.5 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

5.3.6 \$simparam() Function

Standard types supported by SIMetrix:

“gdev”

“gmin”

“simulatorSubversion”

“simulatorVersion”

“sourceScaleFactor” - includes pseudo transient scale factor

“tnom”

Additional SIMetrix extensions:

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

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

5.3.7 \$fopen() Function

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

5.4 Verilog-A Interaction with SIMetrix Features

5.4.1 Real-Time Noise

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

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

5.4.2 Transient Snapshots

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

5.4.3 Pseudo-Transient Analysis

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

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

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

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

Copyright © SIMetrix Technologies Ltd. 1992-2019

SIMetrix 8.3 Verilog A Manual