Verilog-A

Language Reference Manual

Analog Extensions to Verilog HDL

Version 1.0

August 1, 1996

Open Verilog International

No part of this work covered by the copyright hereon may be reproduced or used in any form or by any means --- graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems --- without the prior written approval of Open Verilog International.

Additional copies of this manual may be purchased by contacting Open Verilog International at the address shown below.

Notices

The information contained in this draft manual represents the definition of the Verilog-A hardware description language as proposed by OVI (Analog TSC) as of January, 1996. Open Verilog International makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this draft manual to a user's requirements. This language is not yet fully defined and is subject to change. It is suitable for learning how to do analog modeling and as a vehicle for providing feedback to the standards committee. Verilog-A should not be used for production design and development.

Open Verilog International reserves the right to make changes to the Verilog-A hardware description language and this manual at any time without notice.

Open Verilog International does not endorse any particular simulator or other CAE tool that is based on the Verilog-A hardware description language.

Suggestions for improvements to the Verilog hardware description language and/or to this manual are welcome. They should be sent to the address below.

Information about Open Verilog International and membership enrollment can be obtained by inquiring at the address below.

Published as: Verilog-A Language Reference Manual
Version 1.0, August 1, 1996.

Published by: Open Verilog International
15466 Los Gatos Blvd., #109071
Los Gatos, CA 95032
Phone: (408) 358-9510
Fax: (408) 358-3910

Printed in the United States of America.

Verilog® is a registered trademark of Cadence Design Systems, Inc.

The following people contributed to the creation, editing and review of this document.

Name	Company	Email
Ramana Aisola	Motorola	aisola@analog-dse.sps.mot.com
Kevin Cameron	Meta-Software	kevinc@metasw.com
Dan FitzPatrick	Apteq	dkf@apteq.com
Vassilios Gerousis	Motorola	gerousis@chdasic.sps.mot.com
Ian Getreu	Analogy	iang@analogy.com
Kim Hailey	Meta Software	kimh@metasw.com
Ken Kundert	Cadence	kundert@cadence.com
Oskar Leuthold	GEC Plessy	leuthold@sv.gpsemi.com
S. Peter Liebmann	Meta Software	peterl@metasw.com
Ira Miller	Motorola	miller@analog-dse.sps.mot.com
Tom Reeder	Viewlogic	treeder@viewlogic.com
Steffen Rochel	Anacad/Mentor	steffen_rochel@mentorg.com
James Spoto	Cadence	spoto@cadence.com
Richard Trihy	Cadence	trihy@cadence.com
Yatin Trivedi	SEVA Technologies	trivedi@seva.com
Alex Zamfirescu	Veribest	a.zamfirescu@ieee.org

Table of Contents

Verilog-A HDL Overview

Overview 1-1
Systems 1-1
Conservative systems 1-2
Kirchhoff's laws 1-3
Signal-flow systems 1-4
Mixed systems 1-5
Natures, disciplines and nodes 1-7
Conventions used in this document 1-8
Contents 1-9

Lexical Tokens

Lexical tokens 2-1
White space 2-1
Comments 2-1
Operators 2-2
Numbers 2-2
Integer constants 2-2
Real constants 2-3
Units for real constants 2-4
Conversion 2-4
Identifiers, keywords, and system names 2-5
Escaped identifiers 2-5
Keywords 2-5

System tasks and functions 2-7
Compiler directives 2-7

Data Types

Integer and real datatypes 3-1
Operators and real numbers 3-2
Conversion 3-2
Parameters 3-2
Type Specification 3-4
Value Range Specification 3-4
Nodes 3-5
Natures 3-5
Disciplines 3-9
Node Declaration 3-11
Implicit Nodes 3-13
Node Compatibility 3-13
Branches 3-15
Branch Declaration 3-15
Accessing Node and branch Signals and Attributes 3-16
Namespace 3-17
Nature and Discipline 3-17
Node 3-17
Branch 3-18

Expressions

Operators 4-1
Operators with real operands 4-2
Binary operator precedence 4-2
Expression evaluation order 4-3
Arithmetic operators 4-4
Relational operators 4-4
Equality operators 4-5
Logical operators 4-5

Bit-wise operators 4-6
Shift operators 4-7
Conditional operator 4-7
Event or 4-7
Built-In Mathematical Functions 4-7
Standard Mathematical Functions 4-8
Transcendental Functions 4-8
Environment Parameters 4-9
Error Handling 4-9
Signal Access Functions 4-9
Analog Operators 4-10
Restrictions on analog operators 4-11
Analog Operators and Tolerances 4-11
Time Derivative Operator 4-11
Time Integral Operator 4-12
Delay Operator 4-13
Transition Filter 4-13
Slew Filter 4-16
Laplace Transform Filters 4-17
Z-Transform Filters 4-19
Limited Exponential 4-22
Analysis Dependent Functions 4-22
Analysis 4-22
AC Stimulus 4-23
Noise 4-24
User defined functions 4-25
Defining a function 4-25
Returning a value from a function 4-27
Calling a function 4-27

Signals

Analog Signals 5-1
Access Functions 5-1
Probes and Sources 5-2

Examples 5-3
Port Branches 5-6
Switch Branches 5-7
Unassigned Sources 5-8
Contribution statements 5-8
Branch Contribution Statements 5-8
Indirect Branch Assignments 5-9

Analog Behavior

Analog procedural block 6-1
Null statement 6-2
Block statement 6-2
Block names 6-3
Procedural assignment 6-3
Conditional statement 6-4
If-else-if Construct 6-5
Case statement 6-5
Constant expression in case statement 6-6
Looping statements 6-6
Generate statement 6-7
Analog events 6-8
Event detection 6-9
Event OR operator 6-9
Global events 6-10
Monitored events 6-11
Announcing Discontinuity 6-13
Time related functions 6-15
Bounding the time step 6-15
Last_Crossing Function 6-16

Hierarchical Structures

Modules 7-1
Top-level modules 7-2

Module instantiation 7-3
Overriding module parameter values 7-5
Defparam statement 7-5
Module instance parameter value assignment by order 7-6
Module instance parameter value assignment by name 7-7
Parameter override precedence 7-7
Parameter dependence 7-8
Ports 7-8
Port association 7-8
Port declarations 7-9
Connecting module ports by ordered list 7-10
Connecting module ports by name 7-11
Port connection rules 7-11
Inheriting Port Natures 7-12
Multi-disciplinary example 7-12
Hierarchical names 7-13
Scope rules 7-15

Scheduling Semantics

Open Issues

Syntax

Keywords

System Tasks and Functions

Compiler Directives

Standard Definitions

Glossary

Index

Section 1

Verilog-A HDL Overview

1.1 Overview

This Verilog-A Hardware Description Language (HDL) language reference manual defines a behavioral language for analog systems. Verilog-A HDL is derived from the IEEE 1364 Verilog HDL specification. This document is intended to cover the definition and semantics of Verilog-A HDL as proposed by Open Verilog International (OVI).

The intent of Verilog-A HDL is to let designers of analog systems and integrated circuits create and use modules that encapsulate high-level behavioral descriptions as well as structural descriptions of systems and components. The behavior of each module can be described mathematically in terms of its terminals and external parameters applied to the module. The structure of each component can be described in terms of interconnected sub-components. These descriptions can be used in many disciplines such as electrical, mechanical, fluid dynamics, and thermodynamics.

Verilog-A HDL is defined to be applicable to both electrical and non-electrical systems description. It supports *conservative* and *signal-flow* descriptions by using the terminology for these descriptions using the concepts of *nodes*, *branches*, and *ports*. The solution of analog behaviors which obey the laws of conservation fall within the generalized form of Kirchhoff's Potential and Flow laws (KPL and KFL). Both of these are defined in terms of the quantities associated with the analog behaviors.

1.2 Systems

A *system* is considered to be a collection of interconnected *components* that are acted upon by a stimulus and produce a response. The components themselves might also be systems, in which case a hierarchical system is defined. If a component does not have any sub-components, then it is considered a primitive component. Each primitive component connects to one or more nodes. The behavior of each component is defined in terms of signal values at each node.

The components connect to nodes through ports to build hierarchy as shown in figure 1-1.

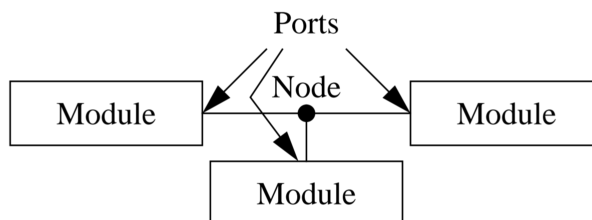


Figure 1-1: Components connect to nodes through ports.

In order to simulate systems, it is necessary to have a complete description of the system and all of its components. Descriptions of systems are given structurally. That is, the description of a system contains instances of components and how they are interconnected. Descriptions of primitive components are given behaviorally. That is, a mathematical description is given that relates the signals at the ports of the component.

1.2.1 Conservative systems

An important characteristic of conservative systems is that there are two values associated with every node (and hence every terminal) - the potential (also known as the across value, or the voltage in electrical systems) and the flow (the through value, or the current in electrical systems). The potential of the node is shared with all terminals connected to the node in such a way that all terminals see the same potential. The flow is shared such that flow from all terminals at a node must sum to zero. In this way, the node acts as an infinitesimal point of interconnection in which the potential is the same everywhere on the node and on which no flow can accumulate. Thus, the node embodies Kirchhoff's Potential and Flow Laws (KPL and KFL). When a component connects to a node through a conservative terminal, it may either affect, or be affected by, either the potential at the node, and/or the flow onto the node through the terminal.

With conservative systems it is also useful to define the concept of a branch. A branch is a path of flow between two nodes through a component. Every branch has an associated potential (the potential difference between the two nodes) and flow.

A behavioral description of a conservative component is constructed as a collection of interconnected branches. The constitutive equations of the component are formulated as to relate the branch potentials and flows. In the probe/source approach, the branch potential or flow is specified as a function of branch potentials and flows. If the branch potential and flow are left unspecified, then the branch acts as a probe. In this case, if the branch flow is used in an expression, the branch potential is forced to zero. Otherwise the branch flow is assumed to be zero and the branch potential is available for use in an expression. Using both the potential and flow of a 'probe' branch in an expression is not

allowed. Nor is specifying both the branch potential and flow at the same time. (While these last two conditions are not really necessary, they do eliminate conditions that are useless and confusing.)

1.2.1.1 Reference nodes

The potential of a single node is given with respect to a reference node. The potential of the reference node, which is called *ground* in electrical systems, is always zero.

1.2.1.2 Reference directions

The reference directions for a generic branch are as follows.

Figure 1-2: Reference directions

The reference direction for a potential is indicated by the plus and minus symbols near each terminal. Given the chosen reference direction, the branch potential is positive whenever the potential of the terminal marked with a plus sign (A) is larger than the potential of the terminal marked with a minus sign (B). Similarly, the flow is positive whenever it moves in the direction of the arrow (in this case from + to -).

Verilog-A HDL uses associated reference directions. A positive flow enters a branch through the terminal marked with the plus sign and exits the branch through the terminal marked with the minus sign.

1.2.2 Kirchhoff's laws

In formulating system equations, Verilog-A HDL uses two sets of relationships. The first are the constitutive relationships that describe the behavior of each component. Constitutive relationships can be kept inside the simulator as built-in primitives, or they can be provided by Verilog-A HDL module definitions.

The second set of relationships, interconnection relationships, describe the structure of the network. Interconnection relationships, which contain information on how the components are connected to each other, are only a function of the system topology. They are independent of the nature of the components.

The Verilog-A HDL simulator uses Kirchhoff's laws to define the relationships between the nodes and the branches. Kirchhoff's laws are typically associated with electrical circuits that relate voltages and currents. However, by generalizing the concepts of

voltages and currents to potentials and flows, Kirchhoff's laws can be used to formulate interconnection relationships for any type of system.

Kirchhoff's laws provide the following properties relating the quantities present on nodes and branches.

- Kirchhoff's Flow Law (KFL)
  The algebraic sum of all flows out of a node at any instant is zero.
- Kirchhoff's Potential Law (KPL)
  The algebraic sum of all the branch potentials around a loop at any instant is zero.

.

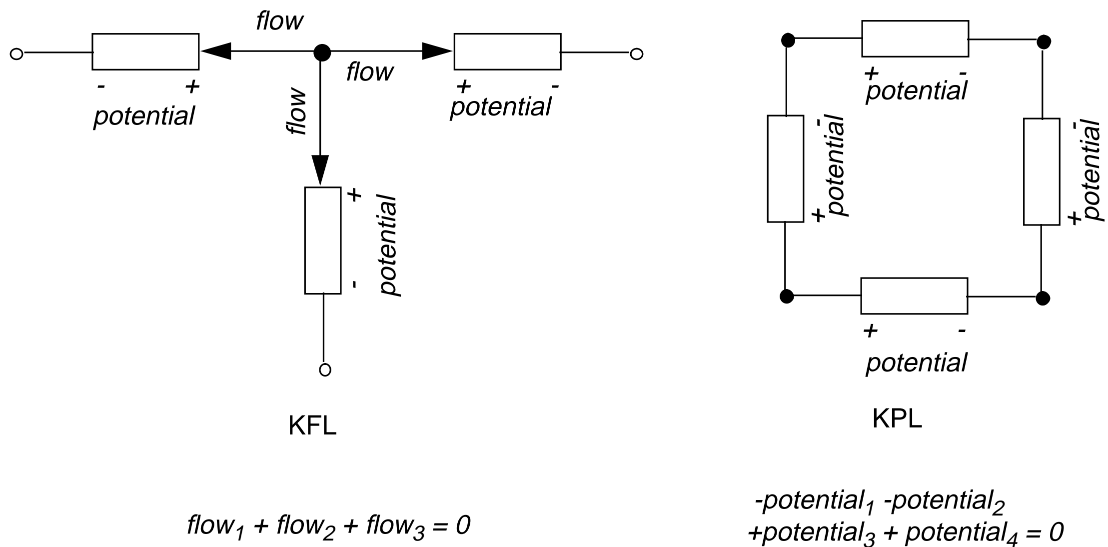


Figure 1-3: Kirchhoff's Flow Law (KFL) and Potential Law (KPL)

These laws imply that a node is infinitely small so that there is negligible difference in potential between any two points on the node and a negligible accumulation of flow.

1.2.3 Signal-flow systems

Unlike conservative systems, signal-flow systems only have one potential associated with every node. As a result, a signal-flow terminal must be unidirectional. It may either read the potential of the node, or it may specify it. Signal-flow terminals are either considered input ports if they pass the potential of the node into a component, or output ports if they specify the potential of a node.

Signal-flow terminals support a subset of the functionality of conservative terminals. As such, one can always use conservative semantics to represent signal-flow components.

There are, however, two important benefits that result from allowing direct description of signal-flow components using signal-flow semantics. First, one only need declare the types of signals that one intends to use. Conversely, one need not declare the types of signals that are not used and therefore for which one would have no basis upon which to make a choice of what the signal type should be. Second, signal-flow semantics require a smaller number of equations and unknowns, and so results in a formulation that is more efficient to simulate.

There are some restrictions that are typically present in signal-flow formulations. For example,

- Typically, one cannot directly interface signal-flow and conservative components.
- Typically, signals are potential-like, making it difficult to represent flow-like signals.
- Typically, components descriptions can only be written in terms of ground-referred signals, making it difficult to write descriptions of components that use floating or differential signals.

1.2.4 Mixed systems

When practicing the top-down design style, it is extremely useful to mix conservative and signal-flow components in the same system. Users typically use signal-flow models early in the design cycle when the system is described in abstract terms, and gradually convert component models to conservative form as the design progresses. Thus, it is important to be able to initially describe a component using a signal-flow model, and later convert it to a conservative model, with a minimum of fuss. It is also important to allow conservative and signal-flow components to be arbitrarily mixed in the same system.

The approach taken is to write component descriptions using conservative semantics, except that terminal and node declarations will only require types for those values that are actually used in the description. Thus, signal-flow terminals will only require the type of one potential to be specified (typically the potential, but could alternatively be the flow), whereas conservative terminals would require types for both values (the potential and flow). For example, consider a differential voltage amplifier, a differential current amplifier, and a resistor. The amplifiers are written using signal-flow terminals and the resistor uses conservative terminals. These examples are meant to illustrate conceptual points only, and are not complete descriptions of the model.

```
module voltage_amplifier (out, in) ;
input in ;
output out ;
voltage  out ,    // Discipline voltage defined elsewhere
         in ;     // with potential access function V()
parameter real GAIN_V = 10.0 ;

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

endmodule
```

In this case, only the voltage on the terminals are declared because only voltage is used in the body of the model.

```
module current_amplifier (out, in) ;
input in ;
output out ;
current  out ,    // Discipline current defined elsewhere
         in ;     // with flow access function I()
parameter real GAIN_I = 10.0 ;

analog
   I(out) <+ GAIN_I * I(in) ;

endmodule
```

Here, only current is used in the body of the model, so only current need be declared at the terminals.

```
module resistor (a, b) ;
inout a, b ;
electrical a, b ;       // access functions are V() and I()
parameter real R = 1.0 ;

analog
   V(a,b) <+ R * I(a,b) ;

endmodule
```

The description of the resistor relates both the voltage and current on the terminals, so both must be declared.

In summary, only those signals types declared on the terminals are accessible in the body of the model. Conversely, only those signals types used in the body need be declared.

This approach provides all of the power of the conservative formulation for both signal-flow and conservative terminals, without forcing types to be declared for unused signals on signal-flow nodes and terminals. In this way, the first benefit of the traditional signal-flow formulation is provided without the restrictions. The second benefit, that of a smaller, more efficient, set of equations to solve, is provided in a manner that is hidden from the user. The simulator begins by treating all terminals as being conservative, which will allow the connection of signal-flow and conservative terminals. This results in additional unnecessary equations for those nodes that only have signal-flow terminals. This situation can be recognized by the simulator and those equations eliminated.

Thus, this approach to allowing mixed conservative/signal-flow descriptions provides the following benefits:

- Conservative components and signal-flow components can be freely mixed. In addition, signal-flow components can be converted to conservative components, and vice versa, by modifying only the component behavioral description.
- Many of the capabilities of conservative terminals, such as the ability to access flow and the ability to access floating potentials, are available with signal-flow terminals.
- Signal-types only have to be given for potentials and flows if they are accessed in a behavioral description.
- If nodes and terminals are used only in a structural description (only in instance statements), then no signal-types need be specified.

1.2.5 Natures, disciplines and nodes

Verilog-A HDL allows definition of nodes based on disciplines. The disciplines associate potential and flow natures for conservative systems or only potential nature for signal-flow systems. The natures are a collection of attributes, including user defined attributes, that describes the units (meter, gram, newton, etc.), absolute tolerance for convergence, and the names of potential and flow access functions.

The disciplines and natures can be shared by many nodes. The compatibility rules help enforce the legal operations between nodes of different disciplines.

1.3 Conventions used in this document

This document is organized into sections, each of which focuses on some specific area of the language. There are subsections within each section to discuss with individual constructs and concepts. The discussion begins with an introduction and an optional rationale for the construct or the concept, followed by syntax and semantic description, followed by some examples and notes.

The formal syntax of Verilog HDL is described using Backus-Naur Form (BNF). The following conventions are used:

1. Lower case words, some containing embedded underscores, are used to denote syntactic categories, for example:

   ```
   module_declaration
   ```

2. Bold face words are used to denote reserved keywords, operators and punctuation marks as required part of the syntax. For example:

   ```
   module      =          ;
   ```

3. A vertical bar separates alternative items. For example:

   ```
   attribute ::=
      abstol | units | identifier
   ```

4. Square brackets enclose optional items. For example:

   ```
   input_declaration ::= input [range]  list_of_ports ;
   ```

5. Braces enclose a repeated item unless it appears in bold face, in which case it stands for itself. The item may appear zero or more times; the repetitions occur from left to right as with an equivalent left-recursive rule. Thus, the following two rules are equivalent:

   ```
   list_of_port_def ::= port_def { , port_def }

   list_of_port_def ::=
        port_def
      | list_of_port_def , port_def
   ```

6. If the name of any category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, *msb*_constant_expression and *lsb*_constant_expression are equivalent to constant_expression, and *node*_identifier is an identifier that is used to identify (declare or reference) a node.

The main text uses *italicized* font when a term is being defined, and `constant-width` font for examples, file names, and while referring to constants.

1.4 Contents

This document contains the following chapters:

1. Verilog-A HDL Overview
   This section gives the overview of analog modeling, basic concepts, and describes Kirchhoff's Potential and Flow Laws.

2. Lexical Tokens
   This section lexical tokens used in Verilog-A HDL.

3. Data Types
   This section describes the data types - integer, real, parameter, nature, discipline, and node - as used in Verilog-A HDL descriptions.

4. Expressions
   This section describes expressions, mathematical functions, and time domain functions used in Verilog-A HDL.

5. Signals
   This section describes signals and branches, access to signals and branches, and various transformation functions.

6. Analog Behavior
   This section describes the basic analog block and procedural language constructs available in Verilog-A HDL for behavioral modeling.

7. Hierarchical Structures
   This section describes how to build hierarchical descriptions using Verilog-A HDL.

A. Scheduling Semantics
   This annex describes the basic simulation cycle as applicable to Verilog-A HDL.

B. Open Issues
   This annex lists the open issues known to the working group.

C. Syntax
   This annex describes formal syntax for all Verilog-A HDL constructs in Bachus-Naur Form (BNF).

D. Keywords
   This annex lists all the words that are recognized in Verilog-A HDL as keywords.

E. System Tasks and Functions
   This annex describes all system tasks and functions in Verilog-A HDL.

F. Compiler Directives
   This annex describes all compiler directives in Verilog-A HDL.

G. Standard Definitions
This annex provides definitions of several natures, disciplines and constants useful writing models in Verilog-A HDL.

H. Glossary
This annex describes various terms used in this document.

Section 2
Lexical Tokens

This section describes the lexical tokens used in Verilog HDL source text and their conventions.

2.1 Lexical tokens

Verilog HDL source text files is a stream of lexical tokens. A *lexical token* consists of one or more characters. The layout of tokens in a source file is free format—that is, spaces and newlines are not syntactically significant other than being token separators, except escaped identifiers (Section 2.6.1).

The types of lexical tokens in the language are as follows:

- white space
- comment
- operator
- number
- string
- identifier
- keyword

2.2 White space

White space contain the characters for spaces, tabs, newlines, and formfeeds. These characters are ignored except when they serve to separate other lexical tokens.

2.3 Comments

The Verilog HDL has two forms to introduce comments. A *one-line comment* starts with the two characters `//` and ends with a newline. A *block comment* starts with `/*` and ends with `*/`. Block comments can not be nested. The one-line comment token `//` does not have any special meaning in a block comment.

2.4 Operators

Operators are single, double, or triple character sequences and are used in expressions. Section 4 discusses the use of operators in expressions.

Unary operators appear to the left of their operand. *Binary operators* appear between their operands. A *conditional operator* has two operator characters that separate three operands.

2.5 Numbers

Constant numbers can be specified as integer constants or real constants. The syntax for constants is as shown below:

```
number ::=
     decimal_number
   | real_number

decimal_number ::=
     [ sign ] unsigned_num

real_number ::=
     [ sign ] unsigned_num . unsigned_num
   | [ sign ] unsigned_num [ . unsigned_num ] e [ sign ] unsigned_num
   | [ sign ] unsigned_num [ . unsigned_num ] E [ sign ] unsigned_num
   | [ sign ] unsigned_num [ . unsigned_num ] unit_letter

sign ::=
   + | -

unsigned_num ::=
   decimal_digit { _ | decimal_digit }

decimal_digit ::=
   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unit_letter ::=
   T | G | M | K | m | u | n | p | f | a
```

Figure 2-1: Syntax for integer and real constants

2.5.1 Integer constants

Integer constants are specified in decimal format as a sequence of digits `0` through `9`, optionally starting with a plus or minus unary operator. The underscore character (_) is legal anywhere in a decimal number except as the first character. The underscore

character is ignored. This feature can be used to break up long numbers for readability purposes.

Examples:

```
27_195_000    // same as 27195000
-659
```

2.5.2 Real constants

The *real constant numbers* are represented as described by IEEE STD-754-1985, an IEEE standard for double precision floating point numbers.

Real numbers can be specified in either decimal notation (for example, 14.72) or in scientific notation (for example, 39e8, which indicates 39 multiplied by 10 to the 8th power). Real numbers expressed with a decimal point have at least one digit on each side of the decimal point.

Examples:

```
1.2
0.1
2394.26331
1.2E12  (the exponent symbol can be e or E)
1.30e-2
0.1e-0
23E10
29E-2
236.123_763_e-12   (underscores are ignored)
```

The following are invalid forms of real numbers because they do not have at least one digit on each side of the decimal point:

```
.12
9.
4.E3
.2e-7
```

2.5.3 Units for real constants

The floating-point numbers can be specified with the following letter symbols for the scale factors indicated.

	$m = 10^{-3}$
$T = 10^{12}$	$u = 10^{-6}$
$G = 10^{9}$	$n = 10^{-9}$
$M = 10^{6}$	$p = 10^{-12}$
$K = 10^{3}$	$f = 10^{-15}$
	$a = 10^{-18}$

Figure 2-2: Symbols used as multipliers to numbers

No space is permitted between the number and the symbol.

This form of floating-point number specification is provided in Verilog-A HDL in addition to the two methods for writing floating-point numbers described earlier.

Example:

Short form	Expanded form
1.3u	1.3e-6 or 0.0000013
5.46K	5460

2.5.4 Conversion

Real numbers are converted to integers by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion take place when a real number is assigned to an integer. The ties are rounded away from zero.

Examples:

```
The real numbers 35.7 and 35.5 both become 36 when
converted to an integer and 35.2 becomes 35.

Converting -1.5 to integer yields -2, converting 1.5 to
integer yields 2.
```

2.6 Identifiers, keywords, and system names

An *identifier* is used to give an object a unique name so it can be referenced. An identifier can be any sequence of letters, digits, dollar signs (`$`), and the underscore characters (_).

The first character of an identifier can not be a digit or `$`; it can be a letter or an underscore. Identifiers are case sensitive.

Examples:

```
shiftreg_a
busa_index
error_condition
merge_ab
_bus3
n$657
```

2.6.1 Escaped identifiers

Escaped identifiers start with the backslash character (\) and end with white space (space, tab, newline). They provide a means of including any of the printable ASCII characters in an identifier (the decimal values 33 through 126, or 21 through 7E in hexadecimal).

Neither the leading back-slash character nor the terminating white space is considered to be part of the identifier. Therefore, an escaped identifier `\cpu3` is treated the same as a non-escaped identifier `cpu3`.

Examples:

```
\busa+index
\-clock
\***error-condition***
\net1/\net2
\{a,b}
\a*(b+c)
```

2.6.2 Keywords

Keywords are predefined non-escaped identifiers that are used to define the language constructs. A Verilog HDL keyword preceded by an escape character is not interpreted as a keyword.

All keywords are defined in lowercase only. Annex D gives a list of all defined keywords.

2.6.2.1 Verilog-A Keywords

In addition to the keywords within Verilog HDL, the following are additional keywords used by Verilog-A HDL.

abstol	discipline	from	nature
access	enddiscipline	generate	potential
analog	endnature	ground	units
branch	exclude	idt_nature	
ddt_nature	flow	inf	

Figure 2-3: List of additional keywords

2.6.2.2 Math Function Keywords

The following are reserved keywords used by the math library.

abs	asin	atanh	cosh	ln	min	sinh	tanh
acos	asinh	atan2	exp	log	pow	sqrt	
acosh	atan	cos	hypot	max	sin	tan	

Figure 2-4: List of keywords for math library

2.6.2.3 Built-in functions

The following are reserved keywords for all built-in functions. The functions are described later in appropriate sections of this document.

ac_stim	delay	initial_step	last_crossing	white_noise
analysis	discontinuity	laplace_nd	noise_table	zi_nd
bound_step	final_step	laplace_np	slew	zi_np
cross	flicker_noise	laplace_zd	timer	zi_zd
ddt	idt	laplace_zp	transition	zi_zp

Figure 2-5: List of built-in functions

2.6.3 System tasks and functions

The **$** character introduces a language construct that enables development of user-defined tasks and functions. A name following the **$** is interpreted as a *system task* or a *system function*.

The syntax for a system task or function is as follows:

```
system_task_or_function ::=
     $system_task_identifier [ ( list_of_arguments ) ] ;
   | $system_function_identifier [ ( list_of_arguments ) ] ;

list_of_arguments ::=
     argument { , [ argument ] }

argument ::=
     expression
```

Figure 2-6: : Syntax for system tasks and functions

Any valid identifier, including keywords already in use in contexts other than this construct can be used as a system task or function name.

Examples:

```
$display ("display a message");
$finish;
```

2.6.4 Compiler directives

The **`** character (the ASCII value 60, called open quote or accent grave) introduces a language construct used to implement compiler directives. The compiler behavior dictated by a compiler directive takes effect as soon as the compiler reads the directive. The directive remains in effect for the rest of the compilation unless a different compiler directive specifies otherwise. A compiler directive in one description file can therefore control compilation behavior in multiple description files.

Any valid identifier, including keywords already in use in contexts other than this construct can be used as a compiler directive name.

Example:

```
`define wordsize 8
```

Section 3
Data Types

Verilog-A HDL supports integer, real, and parameter data types as found in Verilog HDL. It also modifies the parameter data types and introduces array of real as an extension of real data type.

Verilog-A HDL introduces a new data type, called node, for representing analog signals. The nodes have disciplines that define the natures of potential and flow and associated attributes.

3.1 Integer and real datatypes

The syntax for declaring **integer** and **real** is as follows:

```
integer_declaration ::=
     integer  list_of_identifiers ;

real_declaration ::=
     real list_of_identifiers ;

list_of_identifiers ::=
   var_name { , var_name }

var_name ::=
     variable_identifier
   | array_identifier [ range ]

range ::=
     upper_limit_constant_expression : lower_limit_constant_expression
```

Figure 3-1: Syntax for integer and real declarations

An *integer* declaration declares one or more variables of type integer. These variables can hold values ranging from -2^{31} to $2^{31}-1$. Array of integers can be declared using a range that defines the upper and lower indices of the array. Both indices must be constant expressions and must evaluate to a positive integer, a negative integer, or zero.

Arithmetic operations performed on integer variables produce 2's complement results.

A *real* declaration declares one or more variables of type real. The real variables are stored as 64 bit quantities, and store the real values as described by IEEE STD-754-1985.

Array of real can be declared using a range that defines the upper and lower indices of the array. Both indices must be constant expressions and must evaluate to a positive integer, a negative integer, or zero.

Both integer and real variables are initialized to zero at the start of the simulation.

Examples:

```
integer a[1:64];              // an array of 64 integer values
real float ;                  // a variable to store real value
real gain_factor[1:30] ;// array of 30 gain multipliers
                        // with floating point values
```

3.1.1 Operators and real numbers

The result of using logical or relational operators on real numbers and real variables is a single-bit scalar value. Not all Verilog-A HDL operators can be used with expressions involving real numbers and real variables.

3.1.2 Conversion

The value of a real variable is converted to an integer by rounding the real number to the nearest integer, rather than by truncating it. Implicit conversion takes place when a real number is assigned to an integer. The ties are rounded away from zero.

3.2 Parameters

The syntax for parameter declarations is as follows:

```
parameter_declaration ::=
      parameter [opt_type] list_of_param_assignments ;

opt_type ::=
      real
    | integer

list_of_param_assignments ::=
      declarator_init
    | list_of_param_assignments , declarator_init

declarator_init ::=
      parameter_identifier = constant_expression opt_range*

opt_range ::=
      from range_specifier
    | exclude range_specifier
    | exclude constant_expression

range_specifier ::=
      start_paren expression1 : expression2 end_paren

start_paren ::=
      [
    | (

end_paren ::=
      ]
    | )

expression1 ::=
      constant_expression | -inf

expression2 ::=
      constant_expression | inf
```

Figure 3-2: : Syntax for parameter declaration

The list of parameter assignments must be a comma-separated list of assignments, where the right hand side of the assignment must be a constant expression, that is, an expression containing only constant numbers and previously defined parameters.

Parameters represent constants, hence it is illegal to modify their value at runtime. However, parameters can be modified at compilation time to have values that are different from those specified in the declaration assignment. This allows customization of module instances. A parameter can be modified with the **defparam** statement, or in the module instance statement.

By nature, analog behavioral specifications are characterized more extensively in terms of parameters than their digital counterparts. There are two fundamental extensions to parameter declarations:

- An optional type for the parameter can be specified in Verilog-A HDL. In IEEE 1364, the type of a parameter defaults to the type of the default expression.
- A range of permissible values can be defined for each parameter. In IEEE 1364, this check had to be done in user model or was left as an implementation specific detail.

3.2.1 Type Specification

The parameter declaration can contain an optional type specification. In this sense, the parameter keyword acts more as a type qualifier than a type specifier. A default value for the parameter must be specified.

The following examples illustrate this concept:

```
parameter real slew_rate = 1e-3 ;
parameter integer size = 16 ;
```

If the type of a parameter is not specified, it is derived from the type of the value of the constant expression.

If the type of the parameter is specified, and the value assigned to the parameter conflicts with the type of the parameter, the value is coerced to the type of the parameter. For example,

```
parameter real size = 10 ;
```

Here, `size` will be coerced to `10.0`.

3.2.2 Value Range Specification

The parameter declaration can contain optional specifications of the permissible range of the values of a parameter. More than one range may be specified for inclusion or exclusion of values as legal values for the parameter.

The use of brackets, `[` and `]`, indicate inclusion of the end points in the valid range. The use of parenthesis, `(` and `)`, indicate exclusion of the end points from the valid range. It is possible to include one end point and not the other using `[ )` and `( ]`. The first expression in the range must be numerically smaller than the second expression in the range

For example,

```
parameter real neg_rail = -15 from [-50:0) ;
parameter integer pos_rail = 15 from (0:50) ;
parameter real gain = 1 from [1:1000] ;
```

Here, the parameter `neg_rail` is given a default value of `-15` and only allowed to acquire values within the range of `-50 <= neg_rail < 0`. Similarly, for parameter `pos_rail`, the default value is `15` and it is only allowed to acquire values within the range

of `0 < pos_rail < 50`. For parameter *gain*, the default value is 1 and it is allowed to acquire values within the range of `1<= gain <= 1000`.

The keyword **`inf`** may be used to indicate infinity. If preceded by a negative sign, it indicates negative infinity. For example,

```
parameter real val3=0 from [0:inf) exclude (10:20) exclude (30:40];
```

A single value may be excluded from the possible valid values for a parameter. For example,

```
parameter real res = 1.0 exclude 0 ;
```

The value of a parameter is checked against the range only at the model build time (also called compile time or link time for different applications), and is not a runtime assertion check.

3.3 Nodes

In addition to the data types supported by IEEE 1364, for continuous time simulation an additional data type, *node*, is introduced in Verilog-A. The fundamental characteristic of a node data type is that the values of a node are defined by simultaneous solution of equations defined by the instances connected to the *node* using Kirchoff's conservation laws. In general, a node represents a point of physical connections between entities of continuous-time description, obeying conservation-law semantics.

A node is characterized by the *discipline* it follows. For example, all low-voltage nodes have certain common characteristics, all mechanical nodes have certain characteristics, etc. Therefore, a node is always declared as a type of discipline. In this sense, a discipline is a user defined type for declaring a node.

A discipline is characterized by the attributes defined in *natures* for potential and flow.

3.3.1 Natures

The natures are a collection of attributes. In Verilog-A HDL, there are several pre-defined attributes. In addition, user-defined attributes may be declared and assigned constant values in a nature.

The nature declarations are at the same level as discipline and module declarations in the source text. That is, natures are declared at the top level, and nature declarations do not nest inside other nature declarations, discipline declarations, or module declarations.

The syntax for defining a nature is as follows:

```
nature_declaration ::=
      nature      nature_name
         [ nature_descriptions ]
      endnature

nature_name ::=
      nature_identifier
   |  nature_identifier : parent_identifier

parent_identifier ::=
      nature_identifier
   |  discipline_identifier.flow
   |  discipline_identifier.potential

nature_descriptions ::=
      nature_description
   |  nature_description nature_descriptions

nature_description ::=
   attribute = constant_expression ;

attribute ::=
      abstol
   |  access
   |  ddt_nature
   |  idt_nature
   |  units
   |  identifier
```

Figure 3-3: Syntax for nature declaration

A nature must be defined between the keywords **nature** and **endnature**. Each nature definition must have a unique identifier as the name of the nature, and must include all the required attributes.

For example,

```
nature current
   units = "A" ;
   access = I ;
   idt_nature = charge ;
   abstol = 1u ;
endnature

nature voltage
   units = "V" ;
   access = V;
endnature
```

3.3.1.1 Derived Natures

A nature may be derived from an already declared nature. This allows the new nature to have the same attributes as the attributes for the other nature. The new nature is called a *derived nature*, and the existing nature is called a *parent nature*. If a nature is not derived from any other nature, then it is called a *base nature*.

In order to derive a new nature from an existing nature, the new nature name should be followed by a colon (:) and the name of the parent nature in the nature definition.

A derived nature may declare additional attributes, or override values of the attributes already declared in the parent nature, with certain restrictions (as outlined in section 3.3.1.2) for the predefined attributes.

The attributes of the derived nature are accessed in the same manner as accessing attributes of any other nature.

For example,

```
nature ttl_curr
   units = "A" ;
   access = I ;
   abstol = 1u ;
endnature

// An alias

nature ttl_node_curr : ttl_curr
endnature

nature new_curr : ttl_curr      // derived, but different
   abstol = 1m ;    // modified for this nature
   max = 12.3 ;     // new attribute for this nature
endnature
```

3.3.1.2 Attributes

Attributes define the value of certain quantities that characterize the nature. There are five predefined attributes — *abstol*, *access*, *idt_nature*, *ddt_nature*, and *units*. In addition, user defined attributes may be defined in a nature.

Attribute declaration assigns a constant expression to the attribute name.

abstol

The *abstol* attribute provides a tolerance measure (metric) for convergence of potential or flow calculation. It specifies the maximum negligible for signals associated with the nature.

It is an required attribute. The constant expression assigned to it must evaluate to a real value.

access

The *access* attribute identifies the name for the access function. When the nature is used to bind potential, the name is used as an access function for the potential; when the nature is used to bind flow, the name is used as an access function for the flow. The usage of access function is described further in section 4.3.

This attribute is required for all parent natures (base natures). It is illegal for a derived nature to change the access attribute; the derived nature always inherits the access attribute of its parent nature. When specified, the constant expression assigned to it must be an identifier (name, not a string).

idt_nature

The *idt_nature* attribute provides a relationship between a nature and the nature that represents its time integral.

When specified, the constant expression assigned to an idt_nature attribute must be the name (not a string) of a nature that is defined elsewhere. It is possible for a nature to be self referring with respect to its idt_nature attribute. In other words, the value of the idt_nature attribute may be the nature that the attribute itself is associated with.

This attribute is optional. While it is possible to override the parent's value of the idt_nature attribute, the nature specified must be related to the nature used for the idt_nature attribute by the parent.

ddt_nature

The *ddt_nature* attribute provides a relationship between a nature and the nature that represents its time derivative.

When specified, the constant expression assigned to a ddt_nature attribute must be the name (not a string) of a nature that is defined elsewhere. It is possible for a nature to be self referring with respect to its ddt_nature attribute. In other words, the value of the ddt_nature attribute may be the nature that the attribute itself is associated with.

This attribute is optional. While it is possible to override the parent's value of the ddt_nature attribute, the nature specified must be related to the nature used for the ddt_nature attribute by the parent.

units

The *units* attribute provides a binding between the value of the access function and the units for that value.

It is a required attribute for all parent natures. It is illegal for a derived nature to define or change the units attribute; the derived nature always inherits the units attribute of its parent nature.

When specified, the constant expression must be a string.

3.3.1.3 User Defined Attributes

In addition to the predefined attributes listed above, a nature can have other attributes that may be useful for analog modeling. Typical examples include certain maximum and minimum values to define valid range, etc.

A user defined attribute may be declared in the same manner as any of the predefined attributes. The name of the attribute must be unique in the nature being defined, and the value being assigned to the attribute must be constant.

3.3.2 Disciplines

A discipline description consists of binding natures to potential and flow.

The syntax for declaring a discipline is as follows:

```
discipline_declaration ::=
      discipline  discipline_identifier
         [ discipline_descriptions ]
      enddiscipline

discipline_descriptions ::=
      discipline_description
    | discipline_description discipline_descriptions

discipline_description ::=
      nature_binding
    | attr_description

nature_binding ::=
      pot_or_flow nature_identifier ;

attr_description ::=
      pot_or_flow . attribute_identifier = constant_expression ;

pot_or_flow ::=
      potential
    | flow
```

Figure 3-4: Syntax for discipline declaration

A discipline must be defined between the keywords **discipline** and **enddiscipline**. Each discipline must have a unique identifier as the name of the discipline.

The discipline declarations are at the same level as nature and module declarations in the source text. That is, disciplines are declared at the top level, and discipline declarations do not nest inside other discipline declarations, nature declarations, or module declarations.

3.3.2.1 Nature Binding

Each discipline can bind a nature to its potential and flow.

Only the name of the nature is specified in the discipline. The nature binding for potential is specified using the keyword **potential**. The nature binding for flow is specified using the keyword **flow**.

The access function defined in the nature bound to potential is used in the model to describe the signal-flow that obeys Kirchhoff's Potential Law (KPL). This access function is called *potential access function*.

The access function defined in the nature bound to flow is used in the model to describe the signal-flow that obeys Kirchhoff's Flow Law (KFL). This access function is called *flow access function*.

Disciplines with two natures are called conservative disciplines, and the nodes associated with conservative disciplines are called conservative nodes. Conservative disciplines must not have the same nature specified for both the potential and the flow. Disciplines with a single nature are called as signal-flow disciplines, and the nodes with signal-flow disciplines are called signal-flow nodes.

Example:

Conservative discipline:

```
discipline electrical
   potential voltage ;
   flow current ;
enddiscipline
```

Signal-flow disciplines:

```
discipline voltage
   potential voltage ;
enddiscipline

discipline current
   flow current;
enddiscipline
```

3.3.2.2 Empty Disciplines

It is possible to define a discipline with no nature bindings. These are known as empty disciplines, and may be used in structural descriptions when you wish to let the components connected to a node determine which natures are to be used for the node. Verilog-A HDL predefines *wire* as an empty discipline.

3.3.2.3 Deriving Natures from Disciplines

A nature may be derived from the nature bound to potential or flow in a discipline. This allows the new nature to have the same attributes as the attributes for the nature bound to the flow or the potential of the discipline.

If the nature binding to the flow or the potential of a discipline changes, the new nature will automatically inherit the attributes for the changed nature.

In order to derive a new nature from flow or potential of a discipline, the nature declaration should include the discipline name followed by the hierarchical separator (.) and the keyword **flow** or **potential**.

A nature derived from the flow or potential of a discipline may declare additional attributes, or override values of the attributes already declared.

For example,

```
nature ttl_curr
   units = "A" ;
   access = I ;
   abstol = 1u ;
endnature

nature ttl_volt
   units = "V" ;
   access = V;
   abstol = 100u ;
endnature

discipline ttl
   potential ttl_volt ;
   flow ttl_curr ;
   flow.abstol = 10u ;
enddiscipline

nature ttl_node_curr : ttl.flow
endnature        // abstol = 10u as modified in ttl

nature ttl_node_volt : ttl.potential
   abstol = 1m ;    // modified for this nature
   max = 12.3 ;     // new attribute for this nature
endnature
```

3.3.3 Node Declaration

Each node declaration is associated with an already declared discipline or an empty discipline called wire. The following syntax is used for declaring nodes:

```
node_declaration ::=
      discipline_identifier [range] list_of_nodes ;
    | wire [range] list_of_nodes ;

range ::=
      [ msb_expression : lsb_expression ]

list_of_nodes ::=
      node_identifier
    | node_identifier , list_of_nodes

msb_expression ::=
      constant_expression

lsb_expression ::=
      constant_expression
```

Figure 3-5: Syntax for node declaration

The discipline must be defined for a node to be declared of the type of a discipline.

If a range is specified for a node, the node is called a vector node; otherwise it is called a scalar node. A vector node is also called an analog bus. All the operators for scalar nodes also operate on the vector nodes.

Examples:

```
electrical [MSB:LSB] n1 ; // MSB and LSB are parameters
voltage [5:0] n2, n3 ;
magnetic inductor ;
wire [10:1] connector1 ;
```

Nodes represent the abstraction of information about signals. As terminals (ports of a module declared as nodes), nodes represent component interconnections. Nodes can be used in the following situations in a model:

- Nodes declared in the module interface define the terminals to the module (See section 7.3.2)
- Nodes declared within the module scope are used for modeling behavior internal to that module.

A node used for modeling a conservative system must have the discipline with both access functions (potential and flow) defined. For modeling a signal-flow system, the discipline of a node can have only one access function.

Nodes declared with an empty discipline do not have declared natures, so such nodes cannot be used in a behavioral description (because the access functions are not known). However, such nodes can be used in structural descriptions, where they inherit the natures from the ports of the instances of modules that connect to them.

3.3.4 Implicit Nodes

Nodes can be used in a structural descriptions without being declared. In this case, the node is implicitly declared to be a scalar node with the empty discipline wire.

3.4 Node Compatibility

Certain operations can be done on nodes only if the two (or more) nodes are compatible. For example, if an access function has two nodes as arguments, they must be compatible. The nodes are considered compatible if their respective disciplines are compatible. The following rules apply in deciding whether two disciplines are compatible:

Self Rule: A discipline is compatible with itself.

Potential Compatibility Rule: If the natures of the two potential are compatible, and the natures of the two flow are not incompatible then the two disciplines are considered compatible.

Flow Compatibility Rule: If the natures of the two flow are compatible, and the natures of the two potential are not incompatible then the two disciplines are considered compatible.

Nature Compatibility Rule: Two natures are compatible if they both exist and are derived from the same base nature.

Nature Incompatibility Rule: Two natures are not incompatible if they are compatible or if one or both do not exist.

Units Value Rule: All compatible natures must have the same value for the attribute **units**. Since a child nature cannot override a base nature's unit, this rule is always maintained.

Empty Discipline Rule: An empty discipline is compatible with all disciplines.

The following example illustrates these rules:

```
nature voltage
   access = V;
   units = "V";
   abstol = 1uV;
endnature

nature current
   access = I;
   units = "A";
   abstol = 1pA;
endnature

discipline electrical
   potential voltage;
   flow current;
enddiscipline

discipline cmos:electrical
   potential.abstol=1mV;
enddiscipline

discipline sig_flow_v
   potential voltage;
enddiscipline

discipline sig_flow_i
   flow current;
enddiscipline

nature position
   access = X;
   units = "m";
   abstol = 1um;
endnature

nature force
   access = F;
   units = "N";
   abstol = 1nN;
endnature

discipline mechanical
   potential position;
   flow force;
enddiscipline

discipline sig_flow_x
   potential position;
enddiscipline

discipline sig_flow_f
   flow force;
enddiscipline

discipline wire
enddiscipline
```

The following compatibility observations can be made from the above example:

- electrical and cmos are compatible disciplines because natures for both potential and flow exist and are derived from the same base natures.
- electrical and sig_flow_v are compatible disciplines because nature for potential is same for both disciplines and nature for flow does not exist in sig_flow_v.
- electrical and sig_flow_i are compatible disciplines because nature for flow is same for both disciplines and nature for potential does not exist in sig_flow_i.
- electrical and mechanical are incompatible disciplines because natures for both potential and flow are not derived from the same base natures.
- electrical and sig_flow_x are incompatible disciplines because nature for both potential are not derived from the same base nature.
- sig_flow_v and sig_flow_i are compatible disciplines as well as sig_flow_v and sig_flow_f are compatible disciplines because the natures do not conflict (the potential natures do not conflict because only sig_flow_v has a potential nature,

and the flow natures do not conflict because sig_flow_v does not have a flow nature)

- wire is compatible with all other disciplines because it has neither a potential nor a flow nature. Without natures, there can be no conflicting natures.

3.5 Branches

A branch is a path between two nodes. If both nodes are conservative, then the branch is a conservative branch and it defines a branch potential and a branch flow. If one node is a signal-flow node, then the branch is a signal-flow branch and it defines either a branch potential or a branch flow, but not both.

3.5.1 Branch Declaration

Each branch declaration is associated with two nodes from which it derives a discipline. These nodes are referred to as the branch terminals. Only one node need be specified, in which case the second is taken to be ground and the discipline for the branch is taken from the specified node. The disciplines for the nodes specified must be compatible (see section 3.4).

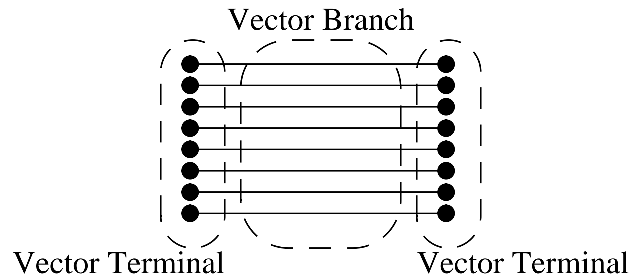
If the same node is specified twice, and if it is a formal node (a port), then the branch is a *port branch*. Specifying an actual node (an internal node) twice on a branch declaration is considered to be an error.

The following syntax is used for declaring branches:

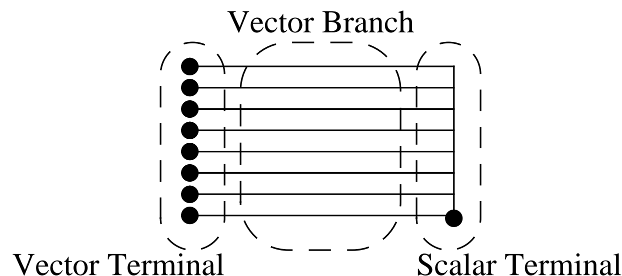
```
branch_declaration ::=
      branch list_of_branches ;

list_of_branches ::=
      list_of_parallel_branches
   |  list_of_parallel_branches , list_of_branches

list_of_parallel_branches ::=
      terminals list_of_branch_identifiers

terminals ::=
      ( node_identifier )
   |  ( node_identifier , node_identifier )

list_of_branch_identifiers ::=
      branch_identifier
   |  branch_identifier , list_of_branch_identifiers
```

Figure 3-6: Syntax for branch declaration

If one of the terminals of a branch is a vector node, then the other terminal must either be a scalar or it must be a vector node of the same size. In this case, the branch is referred to as being a vector branch. When both terminals are vectors, the scalar branches that make up the vector branch connect between the corresponding scalar nodes that make up the vector terminals.



When one terminal is a vector and the other is a scalar, there is one scalar branch connecting to each scalar node in the vector terminal, and the other terminal of each branch connects to the scalar terminal.



3.5.2 Accessing Node and branch Signals and Attributes

Signals on nodes and branches can be accessed only by the access functions of the discipline associated with them. The name of the node or the branch must be specified as the argument to the access function.

For example,

```
electrical out, in ;      // as defined in Section 3.3.2.1
parameter real gm = 1 ;

analog
   I(out) <+ gm*V(in) ;


electrical p, n;
branch (p,n) res;
parameter real R = 50;

analog
   V(res) <+ R*I(res);
```

The attributes are attached to the nature of potential or flow. Therefore, the attributes for a node or a branch can be accessed using the hierarchical referencing operator (.) to the potential or flow for the node or the branch.

For example,

```
electrical a, b, n1, n2;
branch (n1, n2) cap ;
parameter real c= 1p;

analog
   I(a,b) <+ c*ddt(V(a,b), a.potential.abstol);

analog
   I(cap) <+ c*ddt(V(cap), cap.potential.abstol) ;
```

The formal syntax for referencing access functions and attributes is as follows:

```
access_function_reference ::=
      access_function_identifier ( node_args )

node_args ::=
      node_identifier
   |  node_identifier , node_identifier

attribute_reference ::=
      node_identifier . pot_or_flow . attribute_identifier
   |  branch_identifier . pot_or_flow . attribute_identifier
```

Figure 3-7: Syntax for referencing access functions and attributes of a node

3.6 Namespace

3.6.1 Nature and Discipline

The natures and disciplines are defined at the same level of scope as that of modules. Thus, identifiers defined as natures or disciplines have the global scope, and allows declaration of nodes inside any module in the same manner as an instance of a module.

3.6.2 Node

The scope rules for node identifiers are the same as the scope rules for any other identifier declarations with one exception - nodes may not be declared anywhere other than the port of a module or in the module itself. In other words, a node may not be declared inside any block (named or unnamed) other than a module; there is no local declaration for a node.

All access functions are always uniquely defined for each node based on the discipline of the node. Each access function is always used with the name of the node as its argument, and a node is always accessed only through its access functions.

The hierarchical reference character (.) may be used to reference a node across the module boundary using the rules specified in IEEE 1364.

3.6.3 Branch

The scope rules for branch identifiers are the same as the scope rules for node identifiers. In other words, branches are declared inside modules but may not be declared inside any block (named or unnamed) other than a module; there is no local declaration for a branch.

The access functions are always uniquely defined for each branch based on the discipline of the branch. The access function is always used with the name of the branch as its argument, and a branch is always accessed only through its access functions.

Section 4
Expressions

This section describes the operators and operands available in the Verilog-A HDL, and how to use them to form expressions. These operators and operands are a subset of those in Verilog HDL because Verilog-A HDL does not support `reg` or other data types with unknown or strength values.

An *expression* is a construct that combines *operands* with *operators* to produce a result that is a function of the values of the operands and the semantic meaning of the operator. Any legal operand, such as an integer or an indexed element from an array of real, without any operator is also considered an expression. Wherever a value is needed in a Verilog-A HDL statement, an expression can be used.

Some statement constructs require an expression to be a *constant expression*. The operands of a constant expression consists of constant numbers and parameter names, but can use any of the operators defined in Table 4-1.

4.1 Operators

The symbols for the Verilog-A HDL operators are similar to those in the C programming language. Table 4-1 lists these operators.

Table 4-1: Operators in Verilog HDL

<table>
<tr><td>+ - * /</td><td>arithmetic</td></tr>
<tr><td>%</td><td>modulus</td></tr>
<tr><td>> >= < <=</td><td>relational</td></tr>
<tr><td>!</td><td>logical negation</td></tr>
<tr><td>&&</td><td>logical and</td></tr>
<tr><td>||</td><td>logical or</td></tr>
<tr><td>==</td><td>logical equality</td></tr>
<tr><td>!=</td><td>logical inequality</td></tr>
<tr><td>~</td><td>bit-wise negation</td></tr>
<tr><td>&</td><td>bit-wise and</td></tr>
<tr><td>|</td><td>bit-wise inclusive or</td></tr>
<tr><td>^</td><td>bit-wise exclusive or</td></tr>
</table>

Table 4-1: Operators in Verilog HDL

^~ or ~^	bit-wise equivalence
<<	left shift
>>	right shift
? :	conditional
or	event or

4.1.1 Operators with real operands

The operators shown in Table 4-2 are legal when applied to real operands. All other operators are considered illegal when used with real operands.

Table 4-2: Legal operators for use in real expressions

<table>
<tr><td>unary +   unary -</td><td>unary operators</td></tr>
<tr><td>+ - * /</td><td>arithmetic</td></tr>
<tr><td>> >= < <=</td><td>relational</td></tr>
<tr><td>! && ||</td><td>logical</td></tr>
<tr><td>==   !=</td><td>logical equality</td></tr>
<tr><td>?:</td><td>conditional</td></tr>
<tr><td>or</td><td>event or</td></tr>
</table>

The result of using logical or relational operators on real numbers is an integer value 0 or 1 (true).

Table 4-2 lists operators that can not be used to operate on real numbers.

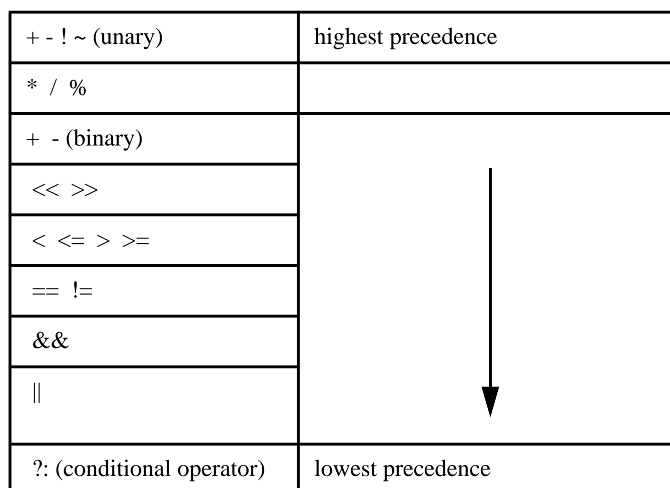
Table 4-3: Operators not allowed for real expressions

%	modulus
<< >>	shift

4.1.2 Binary operator precedence

The precedence order of *binary operators* and the *conditional operator* (**?:**) is shown below in Table 4-4.

Table 4-4: Precedence rules for operators



Operators	Precedence
+ - ! ~ (unary)	highest precedence
* / %	
+ - (binary)	
<< >>	
< <= > >=	
== !=	
&&	
||	
?: (conditional operator)	lowest precedence

Operators shown on the same row in Table 4-4 have the same precedence. Rows are arranged in order of decreasing precedence for the operators. For example, `*`, `/`, and `%` all have the same precedence, which is higher than that of the binary `+` and `-` operators.

All operators associate left to right with the exception of the conditional operator which associate right to left. Associativity refers to the order in which the operators having the same precedence are evaluated. Thus, in the following example `B` is added to `A` and then `C` is subtracted from the result of `A+B`.

```
A + B - C
```

When operators differ in precedence, the operators with higher precedence associate first. In the following example, `B` is divided by `C` (division has higher precedence than addition) and then the result is added to `A`.

```
A + B / C
```

Parentheses can be used to change the operator precedence.

```
(A + B) / C         // not the same as A + B / C
```

4.1.3 Expression evaluation order

The operators follow the associativity rules while evaluating an expression as described in section 4.1.2. However, if the final result of an expression can be determined early, the entire expression need not be evaluated. This is called *short-circuiting* an expression evaluation.

```
integer A, B, C, result ;
result = A & (B | C) ;
```

If `A` is known to be zero, the result of the expression can be determined as zero without evaluating the sub-expression `B | C`.

4.1.4 Arithmetic operators

The binary arithmetic operators are the following:

Table 4-5: Arithmetic operators defined

a + b	a plus b
a - b	a minus b
a * b	a multiply by b
a / b	a divide by b
a % b	a modulo b

The integer division truncates any fractional part toward zero. The modulus operator, for example `y % z`, gives the remainder when the first operand is divided by the second, and thus is zero when `z` divides `y` exactly. The result of a modulus operation takes the sign of the first operand.

The unary arithmetic operators take precedence over the binary operators. The unary operators are the following:

Table 4-6: Unary operators defined

+m	unary plus m (same as m)
−m	unary minus m

Table 4-7 gives examples of modulus operations.

Table 4-7: Examples of modulus operations

Modulus Expression	Result	Comments
10 % 3	1	10/3 yields a remainder of 1
11 % 3	2	11/3 yields a remainder of 2
12 % 3	0	12/3 yields no remainder
-10 % 3	-1	the result takes the sign of the first operand
11 % -3	2	the result takes the sign of the first operand

4.1.5 Relational operators

Table 4-8 lists and defines the relational operators

Table 4-8: The relational operators defined

a < b	a less than b
a > b	a greater than b
a <= b	a less than or equal to b
a >= b	a greater than or equal to b

An expression using these *relational operators* yields the value `0` if the specified relation is *false*, or the value `1` if it is *true*.

All the relational operators have the same precedence. Relational operators have lower precedence than arithmetic operators.

The following examples illustrate the implications of this precedence rule:

```
a < foo - 1         // this expression is the same as
a < (foo - 1)       // this expression, but . . .
foo - (1 < a)       // this one is not the same as
foo - 1 < a         // this expression
```

When `foo - (1 < a)` evaluates, the relational expression evaluates first and then either zero or one is subtracted from `foo`. When `foo - 1 < a` evaluates, the value of `foo` operand is reduced by one and then compared with `a`.

4.1.6 Equality operators

The *equality operators* rank lower in precedence than the relational operators. Table 4-9 lists and defines the equality operators.

Table 4-9: The equality operators defined

a == b	a equal to b,
a != b	a not equal to b,

Both equality operators have the same precedence. These operators compare the value of the operands. As with the relational operators, the result will be `0` if comparison fails, `1` if it succeeds.

4.1.7 Logical operators

The operators *logical and* (`&&`) and *logical or* (`||`) are logical connectives. The result of the evaluation of a logical comparison can be `1` (defined as *true*), or `0` (defined as *false*). The precedence of `&&` is greater than that of `||`, and both are lower than relational and equality operators.

A third logical operator is the unary *logical negation* operator `!`. The negation operator converts a non-zero or true operand into `0` and a zero or false operand into `1`.

The following expression performs a logical and of three sub-expressions without needing any parentheses:

```
a < param1 && b != c && index != lastone
```

However, it is recommended for readability purposes that parentheses be used to show very clearly the precedence intended, as in the following rewrite of the above example:

```
(a < param1) && (b != c) && (index != lastone)
```

4.1.8 Bit-wise operators

The *bit-wise operators* perform bit-wise manipulations on the operands—that is, the operator combines a bit in one operand with its corresponding bit in the other operand to calculate one bit for the result. The logic tables below show the results for each possible calculation.

Table 4-10: Bit-wise binary and operator

&	**0**	**1**
0	0	0
1	0	1

Table 4-11: Bit-wise binary or operator

|	**0**	**1**
0	0	1
1	1	1

Table 4-12: Bit-wise binary exclusive or operator

^	**0**	**1**
0	0	1
1	1	0

Table 4-13: Bit-wise binary exclusive nor operator

^~ ~^	**0**	**1**
0	1	0
1	0	1

Table 4-14: Bit-wise unary negation operator

~	
0	1
1	0

4.1.9 Shift operators

The *shift operators*, `<<` and `>>`, perform left and right shifts of their left operand by the number of bit positions given by the right operand. Both shift operators fill the vacated bit positions with zeroes. The right operand is always treated as an unsigned number.

```
integer start, result;
anlog begin
    start = 1;
    result = (start << 2);
end
```

In this example, the register `result` is assigned the binary value `0100`, which is `0001` shifted to the left two positions and zero filled.

4.1.10 Conditional operator

The *conditional operator*, also known as *ternary operator*, is right associative and must be constructed using three operands separated by two operators with the following syntax:

```
conditional_expression ::=
        expression1 ? expression2 : expression3
```

Figure 4-1: Syntax for conditional operator

The evaluation of a conditional operator begins with the evaluation of expression1. If expression1 evaluates to false (0), then expression3 is evaluated and used as the result of the conditional expression. If expression1 evaluates to true (value other than 0), then expression2 is evaluated and used as the result.

4.1.11 Event or

The event **`or`** operator performs an or of events. See section 6.9.2 for events and triggering of events.

4.2 Built-In Mathematical Functions

Verilog-A HDL supports the following standard mathematical functions.

4.2.1 Standard Mathematical Functions

These are the standard mathematical functions supported by Verilog-A HDL. The operands must be numeric (integer or real). For *min, max*, and *abs*, if either operand is *real*, both are converted to *real*, as is the result. All other arguments are converted to real.

Function	Description	Domain
ln(x)	Natural logarithm	$x > 0$
log(x)	Decimal logarithm	$x > 0$
exp(x)	Exponential	$x < 80$
sqrt(x)	Square root	$x > 0$
min(x, y)	Minimum	All x, all y
max(x, y)	Maximum	All x, all y
abs(x)	Absolute	All x
pow(x, y)	Power. x^y	All x, all y

4.2.2 Transcendental Functions

These are the trigonometric and hyperbolic functions supported by Verilog-A HDL. All operands must be of the numeric type (integer or real) and are converted to real if necessary.

All arguments to the trigonometric and hyperbolic functions are specified in radians.

Function	Description	Domain
sin(x)	Sine	All x
cos(x)	Cosine	All x
tan(x)	Tangent	$x \neq n\left(\frac{\pi}{2}\right)$, n is odd
asin(x)	Arc-sine	$-1 \leq x \leq 1$
acos(x)	Arc-cosine	$-1 \leq x \leq 1$
atan(x)	Arc-tangent	All x
atan2(x,y)	Arc-tangent of x/y	All x, All y
hypot(x,y)	sqrt($x^2 + y^2$)	All x, All y
sinh(x)	Hyperbolic sine	All x

Function	Description	Domain
cosh(x)	Hyperbolic cosine	All x
tanh(x)	Hyperbolic tangent	All x
asinh(x)	Arc-hyperbolic sine	All x
acosh(x)	Arc-hyperbolic cosine	$x \geq 1$
atanh(x)	Arc-hyperbolic tangent	$-1 \leq x \leq 1$

4.2.3 Environment Parameters

These functions return information about the current environment parameters. They take no arguments and return a real number.

Function	Returns
$realtime	Current simulation time in seconds.
$temperature	Ambient temperature in kelvin.
$vt	Thermal voltage (kT/q).
$vt(temp)	Thermal voltage at given temperature.

4.2.4 Error Handling

All math functions not defined for any input must report an error.

4.3 Signal Access Functions

Access functions are used to access signals on nodes, ports, and branches. The name of the access function for a signal is taken from the discipline of the node, port, or branch to which the signal is associated. If the access function is used in an expression, the access function returns the value of the signal. If the access function is being used on the left side of a branch assignment or contribution statement, it assigns a value to the signal. The following table shows how access functions can be applied to branches, nodes, and ports. In this table, *b1* refers to a branch, *n1* and *n2* represent either nodes or ports, and *p1* represents a port. These branches, nodes, and ports are assumed to belong to the

electrical discipline where V is the name of the access function for the voltage (potential), and I is the name of the access function for the current (flow).

Example	Comments
V(b1)	Accesses the voltage across branch *b1*
V(n1)	Accesses the voltage of *n1* (a node or a port) relative to ground
V(n1,n2)	Accesses the voltage difference between *n1* and *n2* (nodes or ports)
V(p1,p1)	Accesses the voltage across the port branch associated with port *p1*
I(b1)	Accesses the current on branch *b1*
I(n1)	Accesses the current flowing from *n1* (a node or port) to ground
I(n1, n2)	Accesses the current flowing between *n1* and *n2*
I(p1,p1)	Accesses the current flow into the module through port *p1*

The argument expression list must be a branch identifier, or a list of one or two node or terminal identifiers. If two node identifiers are given as arguments to an access function, they must not be the same identifier. If two port identifiers are given as arguments, and the identifiers are the same, then the branch defined by the access function is a port branch. The access function name must match the discipline declaration for the nodes, ports, or branch given in the argument expression list. In this case, V and I were used as examples of access functions for electrical potential and flow.

4.4 Analog Operators

Analog operators are functions that operate on more than just the current value of their arguments. Rather, they maintain internal state and their output is a function of both the input and the internal state.

Analog operators operate on an expression and return a value. The expression and the return value may be either a scalar or a vector. If the expression is vector valued, then the return value is vector-valued with the same dimension.

Analog operators are also referred to as filters. They include the time derivative, time integral, and delay operators from calculus. They also include the transition and slew filters, that are used to remove discontinuity from piecewise constant and piecewise continuous waveforms. Finally they include more traditional filters, such as those described with Laplace and Z-transform descriptions.

One special analog operator is the *$limexp()* function, which is a version of the *exp()* function with built-in limits that improves convergence.

4.4.1 Restrictions on analog operators

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

Analog operators must not be used inside conditional statements (*if* and *case*) unless the conditional expression that controls the statement consists of terms that cannot change their value during the course of an analysis. In particular, the conditional expression can only consist of literal numerical constants, parameter values, and the *analysis()* function.

Analog operators are not allowed in the *forever*, *repeat*, *while,* and *for* iteration statements. They are, however, allowed in *generate* statement.

Finally, analog operators can only be used inside an *analog* block. They cannot be used inside a user defined function.

These restrictions are present to prevent use that would cause the internal state to be corrupted or become out-of-date, which results in anomalous behavior.

4.4.2 Analog Operators and Tolerances

Generally, simulators formulate the mathematical description of the system in terms of first-order differential equations and solve them numerically. There is no direct way to solve a set of nonlinear differential equations so iterative approaches are used. When using iterative approaches, one must have criteria used to determine when the algorithm is close enough to the solution to stop the iteration. Tolerances are used for this purpose. Thus, each equation must have a tolerance.

Occasionally, analog operators will require that new equations and new unknowns be introduced by the simulator to convert a module description into a set of first-order differential equations. In this case, the simulator will attempt to determine from context which tolerance should be associated with the new equation and new unknown. Alternatively, these operators allow tolerances to be specified.

Specifying literal values for tolerances can reduce the ability of a module to be reused with signals of much different size. This issue can be avoided by using the *abstol* attribute from nodes and branches.

4.4.3 Time Derivative Operator

The *ddt* operator computes the time derivative of its argument.

Operator	Example	Comments
ddt	ddt(x)	Returns $\frac{d}{dt}x(t)$, the time-derivative of x
	ddt(x, *abstol*)	Same as above, except absolute tolerance is specified explicitly.

In DC analysis, *ddt*() returns zero. The optional parameter *abstol* is used as an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context in which *ddt* is used. See Section 4.4.2 on page -11 for more information. The absolute tolerance applies to the output of the *ddt* operator, and is the largest signal level that is considered negligible.

4.4.4 Time Integral Operator

The *idt* operator computes the time-integral of its argument.

Operator	Example	Comments
idt	idt(x)	Returns $\int_0^t x(\tau)d\tau$, the time-integral of x from 0 to t with the initial condition being computed in the DC analysis.
	idt(x,a)	Returns $\int_0^t x(\tau)d\tau + a$, the time-integral of x from 0 to t with initial condition a. In DC analysis, a is returned.
	idt(x,a,*assert*)	Returns $\int_{t_0}^t x(\tau)d\tau + a$, the time-integral of x from t_0 to t with initial condition a. *Assert* is a integer-valued parameter. *idt* returns a when *assert* is nonzero. t_0 is the time when *assert* last became 0.
	idt(x,a,*assert*,*abstol*)	Same as above, except absolute tolerance is specified explicitly.

When specified with initial conditions, the *idt*() operator returns the value of the initial condition in DC and IC analyses and whenever *assert* is given and is nonzero. Without

initial conditions, *idt* multiplies its argument by infinity in DC analysis. Hence, without initial conditions, it must be used in a system with feedback that forces its argument to zero. The optional parameter *abstol* is used as an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context in which *idt* is used. See section 4.4.2 for more information. The absolute tolerance applies to the input of the *idt* operator and is the largest signal level that is considered negligible.

4.4.5 Delay Operator

Delay implements transport delay for continuous waveforms. (Use the transition function to delay discrete-valued waveforms.) *expression* is delayed by the amount *time_delay*. There are two forms of the delay function, the first does not allow the delay to vary, and the second allows it to vary within a fixed interval. In both cases, *time_delay* must be nonnegative. In the first case, changes to the parameter *time_delay* are ignored and the value initially specified is used. In the second case, *time_delay* can change as long as it remains between 0 and *max_delay*, however, changes to *max_delay* are ignored and the initial value is used.

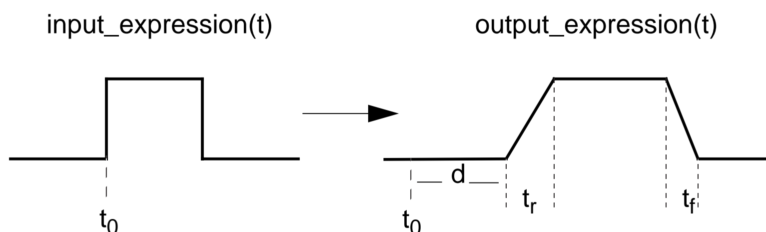
The general forms are

```
delay(expression, time_delay)
delay(expression, time_delay, max_delay)
```

4.4.6 Transition Filter

transition smooths out piece-wise constant waveforms. The transition filter is used to imitate transitions and delays on digital signals. (For non-piecewise-constant signals see *slew*). This function provides controlled transitions between discrete signal levels by setting the rise time and fall time of signal transitions. *transition* stretches instantaneous changes in signals over a finite amount of time, as shown below, and can delay the transitions

.



The general form is

```
transition(expression [ , delay [ , rise_time [ , fall_time]]])
```

transition takes the following arguments (all *real* numbers):

- The input expression

- The delay time (must be nonnegative)
- The rise time (must be positive)
- The fall time (must be positive)

The input expression is expected to evaluate over time to a piecewise constant waveform. When applied, *transition* forces all positive transitions of *expression* to occur over *rise_time* and all negative transitions to occur in *fall_time*, after an initial delay of *delay*. Thus, *delay* models transport delay and *rise_time* and *fall_time* model inertial delay.

transition returns a *real* number that over time describes a piecewise linear waveform. The transition function also causes the simulator to place time-points at both corners of a transition to assure that each transition is adequately resolved. Use short transitions or a short non-zero delay with caution as these can cause the simulator to slow down in order to meet accuracy constraints.

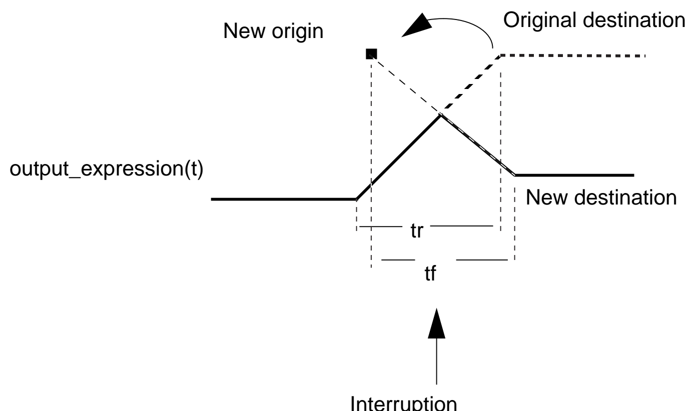
delay, rise_time, and *fall_time* are optional. If *delay* is not specified, it is taken to be zero. If only the *rise_time* value is specified, the simulator uses it for both rise and fall times. If neither rise nor fall time are specified, the rise and fall time are taken to be one unit of time (as defined by the *timescale* complier directive) and no attempt is made to control the time step to follow the trailing corner on the transition. In DC analysis, *transition* passes the value of the *expression* directly to its output.

transition is designed to smooth out piecewise constant waveforms. When applied to waveforms that vary smoothly, the simulation results are generally unsatisfactory. In addition, applying the transition function to a continuously varying waveform can cause the simulator to run slowly. Use *transition* for discrete signals and *slew* for continuous signals.

If interrupted on a rising transition, *transition* tries to complete the transition in the specified time.

- If the new final value level is below the value level at the point of the interruption (the current value), *transition* uses the old destination as the origin.
- If the new destination is above the current level, the first origin is retained.

In the following example, a rising transition is interrupted near its midpoint, and the new destination level of the value is below the current value. For the new origin and destination, *transition* computes the slope that completes the transition from the origin (not the current value) in the specified transition time. It then uses the computed slope to transition from the current value to the new destination.



With larger delays, it is possible for a new transition to be specified before a previously specified transition starts. The transition function handles this by deleting any transitions that would follow a newly scheduled transition. A transition function can have an arbitrary number of transitions pending. A transition function can be used in this way to implement transport delay for discrete-valued signals.

Because the transition function cannot be linearized in general, it is not possible to accurately represent a transition function in AC analysis. The AC transfer function is approximately modeled as having unity transmission for all frequencies in all situations. Because the transition function is intended to handle discrete-valued signals, the small signals present in AC analysis rarely reach transition functions. As a result, the approximation used is generally sufficient.

4.4.6.1 QAM Modulator

In this example, the transition function is used to control the rate of change of the modulation signal in a QAM modulator.

```
module qam16(out, in) ;
parameter freq=1.0, ampl=1.0, delay=0, ttime=1.0/freq ;
input [0:4] in ;
output out ;
electrical in, out ;
real x, y ;
integer row, col ;

analog begin
   row = 2*(V(in[3]) > thresh) + (V(in[2]) > thresh) ;
   col = 2*(V(in[1]) > thresh) + (V(in[0]) > thresh) ;
```

```
      x = transition(row - 1.5, delay, ttime) ;
      y = transition(col - 1.5, delay, ttime) ;
      V(out) <+ ampl * x * cos(2 * 'M_PI * freq * $realtime())
                 + ampl * y * sin(2 * 'M_PI * freq * $realtime()) ;
      bound_step(0.1 / freq) ;
   end
   endmodule
```

4.4.6.2 A-D Converter

The following example, an N-bit analog to digital converter, demonstrates the ability of the transition function to handle vectors.

```
module a2d(in, clk, out) ;
parameter bits=8, fullscale=1.0, delay=0, ttime=10n ;
input in, clk ;
output [0:bits-1] out ;
electrical in, clk, out ;
real sample, thresh ;
integer result[0:bits-1], i ;

analog begin
   @(cross(V(clk)-2.5, +1) begin
      sample = V(in) ;
      thresh = full_scale/2.0 ;
      for (i=bits-1; i>=0; i=i-1) begin
           if (sample > thresh) begin
              result[i] = 1 ;
              sample = sample - thresh ;
           end
           else result[i] = 0 ;
           sample = 2.0*sample ;
      end
   end
   V(out) <+ transition(result,delay,ttime) ;
end
endmodule
```

4.4.7 Slew Filter

The *slew* contribution filter, bounds the rate of change (slope) of the waveform. A typical use for *slew* is generating continuous signals from piecewise continuous signals. (For discrete-valued signals, see *transition*.) The general form is

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

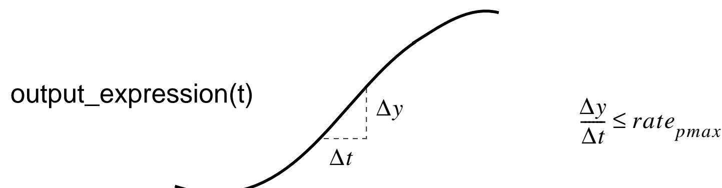
slew takes the following arguments (all *real* numbers):

- The input expression
- The maximum positive slew rate

- The maximum negative slew rate

When applied, *slew* forces all transitions of *expression* faster than *max_pos_slew_rate* to change at *max_pos_slew_rate* rate for positive transitions and limits negative transitions to *max_neg_slew_rate* rate

.



The two rate values are optional. *max_pos_slew_rate* must be greater than 0 and *max_neg_slew_rate* must be less than 0. If only one rate is specified, its absolute value is used for both rates. If no rates are specified, *slew* passes the signal through unchanged. If the rate of change of *expression* is less than the specified maximum slew rates, *slew* returns the value of *expression*. In DC analysis, *slew* simply passes the value of the destination to its output. In AC small-signal analyses, the slew function has unity transfer function except when slewing, in which case it has zero transmission through the function.

4.4.8 Laplace Transform Filters

The Laplace transform filters implement lumped linear continuous-time filters. Each filter takes an optional parameter ε, which is used as an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context in which the filter is used.

4.4.8.1 laplace_zp

laplace_zp implements the zero-pole form of the Laplace transform filter.

```
laplace_zp(expr, ζ, ρ [ , ε ])
```

where ζ (zeta) is a vector of *M* pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, ρ (rho) is the vector of *N* real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is

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

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

4.4.8.2 laplace_zd

laplace_zd implements the zero-denominator form of the Laplace transform filter.

```
laplace_zd(expr, ζ, d [ , ε ])
```

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, d is the vector of N real numbers that contains the coefficients of the denominator. Its transfer function is

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

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

4.4.8.3 laplace_np

laplace_np implements the numerator-pole form of the Laplace transform filter.

```
laplace_np(expr, n, ρ [ , ε ])
```

where n is a vector of M real numbers that contains the coefficients of the numerator. Similarly, ρ (rho) is a vector of N pairs of real numbers. Each pair represents a pole, the first number in the pair is the real part of the pole, and the second is the imaginary part. The transfer function is

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

where n_k is the coefficient of the kth power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the kth pole. If a pole is real, the imaginary part must be

specified as 0. If a pole is complex, its conjugate must also be present. If a pole is zero, then the term associated with it is implemented as s rather than $(1 - s/\rho)$.

4.4.8.4 laplace_nd

laplace_nd implements the numerator-denominator form of the Laplace transform filter.

```
laplace_nd(expr, n, d [ , ε ])
```

where n is an vector of M real numbers that contains the coefficients of the numerator, and d is a vector of N real numbers that contains the coefficients of the denominator. The transfer function is

$$H(s) = \frac{\sum_{k=0}^{M} n_k s^k}{\sum_{k=0}^{N} d_k s^k}$$

where n_k is the coefficient of the kth power of s in the numerator, and d_k is the coefficient of the kth power of s in the denominator.

4.4.8.5 Examples

```
V(out) <+ laplace_zp(V(in), [-1,0], [-1,-1,-1,1]);
```

implements

$$H(s) = \frac{1 + s}{\left(1 + \frac{s}{1 + j}\right)\left(1 + \frac{s}{1 - j}\right)}$$

```
V(out) <+ laplace_nd(V(in), [0,1], [-1,0,1]);
```

implements

$$H(s) = \frac{s}{s^2 - 1}$$

Finally, this example implements a band-limited white noise source with

$$\overline{v_{out}^2} = \frac{k}{|s^2 - 1|^2}$$

```
V(out) <+ laplace_zp(white_noise(k), [], [-1,-1,-1,1]);
```

4.4.9 *Z*-Transform Filters

The *Z*-transform filters implement linear discrete-time filters. Each filter supports the a parameter T that specifies the sampling period of the filter. A filter with unity transfer

function acts like a simple sample-and-hold that samples every T seconds and exhibits no delay.

All Z-transform filters share three common arguments, T, τ, and t_0. T specifies the period of the filter, is mandatory, and it must be positive. τ specifies the transition time, is optional, and must be nonnegative. If the transition time is specified and is nonzero, the timestep is controlled to accurately resolve both the leading and trailing corner of the transition. If it is not specified, the transition time is taken to be one unit of time (as defined by the *timescale* compiler directive) and the timestep is not controlled to resolve the trailing corner of the transition. If the transition time is specified as 0, then the output is abruptly discontinuous. It is not recommended that a *Z*-filter with 0 transition time be directly assigned to a branch. Finally t_0 specifies the time of the first transition, and is also optional. If not given, the first transition occurs at t=0.

4.4.9.1 zi_zp

zi_zp implements the zero-pole form of the *Z* transform filter.

`zi_zp(expr,` ζ, ρ, T [, τ [, t_0]])

where ζ (zeta) is a vector of M pairs of real numbers. Each pair represents a zero, the first number in the pair is the real part of the zero, and the second is the imaginary part. Similarly, ρ (rho) is the vector of N real pairs, one for each pole. The poles are given in the same manner as the zeros. The transfer function is

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

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

4.4.9.2 zi_zd

zi_zd implements the zero-denominator form of the *Z* transform filter.

`zi_zd(expr,` ζ, d, T [, τ [, t_0]])

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

Similarly, d is the vector of N real numbers that contains the coefficients of the denominator. Its transfer function is

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

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

4.4.9.3 zi_np

zi_np implements the numerator-pole form of the Z transform filter.

zi_np(expr, n, ρ, T [, τ [, t_0]])

where n is a vector of M real numbers that contains the coefficients of the numerator. Similarly, ρ (rho) is a vector of N pairs of real numbers. Each pair represents a pole, the first number in the pair is the real part of the pole, and the second is the imaginary part. The transfer function is

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

where n_k is the coefficient of the kth power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the kth pole. If a pole is real, the imaginary part must be specified as 0. If a pole is complex, its conjugate must also be present. If a pole is zero, then the term associated with it is implemented as z rather than $(1 - z/\rho)$.

4.4.9.4 zi_nd

zi_nd implements the numerator-denominator form of the Z transform filter.

```
zi_nd(expr, n, d, T [ , τ [ , t0] ])
```

where *n* is an vector of *M* real numbers that contains the coefficients of the numerator, and *d* is a vector of *N* real numbers that contains the coefficients of the denominator. The transfer function is

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

where n_k is the coefficient of the kth power of *s* in the numerator, and d_k is the coefficient of the kth power of *s* in the denominator.

4.4.10 Limited Exponential

The *$limexp()* function is an operator whose internal state contains information about the argument on previous iterations. It returns a real value that is the exponential of its single real argument, however it internally limits the change of its output from iteration to iteration in order to improve convergence. On any iteration where the change in the output of the *$limexp()* function is bounded, the simulator is prevented from terminating the iteration. Thus, the simulator can only converge when the output of *$limexp()* equals the exponential of the input. The apparent behavior of *$limexp()* is not distinguishable from *exp()*, except using *$limexp()* to model semiconductor junctions generally results in dramatically improved convergence.

4.5 Analysis Dependent Functions

This section describes the *analysis()* function, which is used to determine which type of analysis is being performed. The remaining functions are used to implement small-signal sources. The small-signal source functions only affect the behavior of a module during small-signal analyses. The small-signal analyses provided by SPICE include the AC and noise analyses, but others are possible. When not active, the small-signal source functions return 0.

4.5.1 Analysis

The analysis function takes one or more string arguments and returns 1 if any argument matches the current analysis type. Otherwise it returns 0.

```
analysis( analysis_list )
```

There is no fixed set of analysis types. Each simulator would support its own set. However, simulators should use the following types to represent analyses that are similar to those provided by SPICE.

Name	Analysis Description
"ac"	.AC analysis.
"dc"	.OP or .DC analysis.
"noise"	.NOISE analysis.
"tran"	.TRAN analysis.
"ic"	The initial-condition analysis that proceeds a transient analysis.
"static"	Any equilibrium point calculation, including a DC analysis as well as those that precede another analysis, such as the DC analysis that precedes an AC or noise analysis, or the IC analysis that precedes a transient analysis.
"nodeset"	The phase during an equilibrium point calculation where nodesets are forced.

Any type names unsupported by a simulator are assumed to not be a match.

Using the analysis function, it is possible to have a module behave differently depending on which analysis is being run. For example, it is possible to implement nodesets or initial conditions using the analysis function and switch branches.

```
if (analysis("ic"))
   V(cap) <+ initial_value;
else
   I(cap) <+ ddt(C*V(cap));
```

4.5.2 AC Stimulus

A small-signal analysis computes the steady-state response of a system that has been linearized about its operating point and is driven by a small sinusoid. The sinusoidal stimulus is provided using the *ac_stim()* function.

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

The AC stimulus function returns 0 during large-signal analyses (such as DC and transient) as well as on all small-signal analyses with names different from *analysis_name*. The name of a small-signal analysis is implementation dependent, though it is expected that the name of the equivalent of a SPICE AC analysis will be named "ac", which is the default value of *analysis_name*. When the name of the small-

signal analysis matches *analysis_name*, the source becomes active and models a source with magnitude *mag* and phase *phase*. The default magnitude is 1 and the default phase is 0. Phase is given in radians.

4.5.3 Noise

Several functions are provided to support noise modeling during small-signal analyses. To model large-signal noise during transient analyses, use the *random* function. The noise functions are often referred to as noise sources. There are three noise functions, one models white noise processes, another models *1/f* or flicker noise processes, and the last interpolates a vector to model a process where the spectral density of the noise varies as a piecewise linear function of frequency. The noise functions are only active in small-signal noise analyses, and return 0 otherwise.

4.5.3.1 white_noise

White noise processes are those whose current value is completely uncorrelated with any previous or future values. This implies that their spectral density does not depend on frequency. They are modeled using

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

where *white_noise* generates white noise with a power of *pwr*. For example, the thermal noise of a resistor could be modelled using

```
I(a,b) <+ V(a,b)/R +
                white_noise(4 * `P_K * $temperature/R, "thermal");
```

The optional *name* argument acts as a label for the noise source that is used if the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same name from the same instance of a module are combined in the noise contribution summary.

4.5.3.2 flicker_noise

The second noise function models flicker noise.

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

where *flicker_noise* generates pink noise with a power of *pwr* at 1Hz that varies in proportion to $1/f^{exp}$.

The optional *name* argument acts as a label for the noise source that is used if the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same name from the same instance of a module are combined in the noise contribution summary.

4.5.3.3 noise_table

The last noise function interpolates a vector to model a process where the spectral density of the noise varies as a piecewise linear function of frequency.

```
noise_table(vector [ , name ])
```

where *vector* contains pairs of real numbers, the first number in each pair is the frequency in Hertz, and the second is the power. *noise_table* performs piecewise linear interpolation to compute the power spectral density generated by the function at each frequency.

The optional *name* argument acts as a label for the noise source that is used if the simulator outputs the individual contribution of each noise source to the total output noise. The contributions of noise sources with the same name from the same instance of a module are combined in the noise contribution summary.

4.5.3.4 Noise model for diode

The noise of a junction diode could be modelled as follows:

```
I(a,c) <+ is*(exp(V(a,c) / (n * $vt)) - 1)
         + white_noise(2 * `P_Q * I(a))
         + flicker_noise(kf * pow(abs(I(a)),af), ef);
```

4.5.3.5 Correlated noise

Each noise function generates noise that is uncorrelated with the noise generated by other functions. Perfectly correlated noise is generated by using the output of one noise function for more than one noise source. Partially correlated noise is generated by combining the output of shared and unshared noise functions.

Consider the case where two noise voltages are perfectly correlated:

```
n = white_noise(pwr);
V(a,b) <+ c1 * n;
V(c,d) <+ c2 * n;
```

One can also model partially correlated noise sources:

```
n1 = white_noise(1-corr);
n2 = white_noise(1-corr);
n12 = white_noise(corr);
V(a,b) <+ Kv * (n1 + n12);
I(b,c) <+ Ki * (n2 + n12);
```

4.6 User defined functions

The purpose of a user defined function is to return a value that is to be used in an expression. All functions are defined and used inside a module.

4.6.1 Defining a function

The syntax for defining a function is as follows:

```
function_declaration ::=
   function [ type ] function_identifier  ;
   function_item_declaration { function_item_declaration }
   statement
   endfunction

type ::=
     integer
   | real

function_item_declaration ::=
     input_declaration
   | block_item_declaration

block_item_declaration ::=
     parameter_declaration
   | integer_declaration
   | real_declaration
   | node_declaration
```

Figure 4-2: : Syntax for function declaration

A function declaration begins with the keyword **function**, followed by the type of the return value from the function, followed by the name of the function and a semicolon, and ends with the keyword **endfunction**.

A type must be specified as a **real** or an **integer**. This is the type of the return value from the function. A function must have at least one input declared. The block item declaration can declare the type of the inputs as well as local variables used in the function.

All the variables, including nodes, declared in the module are accessible in the function.

The following example defines a function called `maxV`, which returns potential of the node that is larger in magnitude.

```
function real maxV;
input n1, n2 ;
electrical n1, n2 ;
begin
   // code to compare potential of two nodes
   maxV = (V(n1) > V(n2)) ? V(n1) : V(n2) ;
end
endfunction
```

4.6.2 Returning a value from a function

The function definition implicitly declares a variable, internal to the function, with the same name as the function. This variable has the same type as the type specified in the function declaration. The function definition initializes the return value from the function by assigning the function result to the internal variable with the same name as the function. The following line from above example illustrates this concept:

```
maxV = (V(n1) > V(n2)) ? V(n1) : V(n2) ;
```

A function definition must include an assignment of the function result value to the internal variable that has the same name as the function name.

4.6.3 Calling a function

A function call is an operand within an expression. The function call has the following syntax:

```
function_call  ::=
   function_identifier ( expression { , expression } )
```

Figure 4-3: : Syntax for function call

The order of evaluation of the arguments to a function call is undefined. A function may not call itself directly or indirectly, that is, recursive functions are not permitted.

The following example uses `maxV` function defined in section 4.6.1

```
V(c29) = maxV(c36, c1) ;
```

Section 5
Signals

5.1 Analog Signals

Analog signals are distinguished from digital signals in that they are derived from *disciplines*. Disciplines are a named set of properties that describe an analog signal. Disciplines, nodes and branches are described in Section 3, and ports are described in Section 7.

This section describes signal access mechanisms and operators in Verilog-A HDL.

5.1.1 Access Functions

Signals on nodes, ports, and branches are accessed using *access functions*. The name of the access function is taken from the discipline of the node, port, or branch associated with the signal.

For example, consider a named electrical branch *b* where *electrical* is a discipline with *V* as the access function for the potential and *I* as the access function for the flow. The potential (voltage) would be accessed with:

```
V(b)
```

and the flow (current) is accessed with

```
I(b)
```

Unnamed branches are accessed in a similar manner, except that the access functions are applied to nodes or ports rather than branches. For example, if *n1* and *n2* are electrical nodes or ports, then

```
V(n1,n2)
```

accesses the potential on the unnamed branch from *n1* to *n2* and

```
V(n1)
```

accesses the potential on the unnamed branch from *n1* to ground. In other words, accessing the potential from a node or port to a node or port defines an unnamed branch. Accessing the potential on a single node or port defines an unnamed branch from that node or port to ground. There can only be one unnamed branch between any two nodes or ports.

An analogous access method is used for flows.

```
I(n1,n2)
```

accesses the flow on the unnamed branch from *n1* to *n2*.

```
I(n1)
```

accesses the flow on the unnamed branch from *n1* to ground.

Thus, accessing the flow from a node or port to a node or port defines an unnamed branch. Accessing the potential on a single node or port defines an unnamed branch from that node or port to ground.

5.1.2 Probes and Sources

It is possible to interpret the behavioral descriptions in Verilog-A HDL as a network of probes and controlled sources. While it is not necessary to do so, it is often helpful for two reasons,

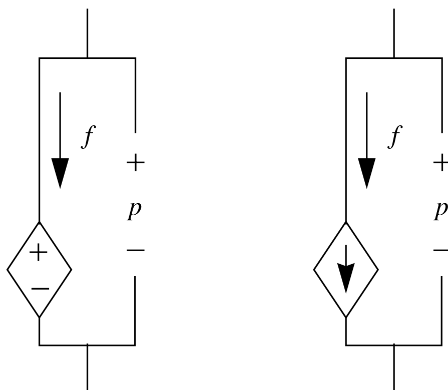
- Describe the component with a network of probes and controlled sources, and then use the simple rules presented here to map the network into a behavioral description.
- Often behavioral descriptions that are difficult to decipher can be more easily understood if it is first converted into a network of probes and controlled sources.

One additional benefit of the probe/source interpretation is that it provides an unambiguous way of defining the behavior for manipulating signals.

5.1.2.1 Sources

A branch, either named or unnamed, is a *source branch* if either the potential or the flow is assigned a value by a contribution statement anywhere in the module. It is a *potential source* if the branch potential is specified, and it is a *flow source* if the branch flow is specified. A branch cannot simultaneously be both a potential and a flow source, though it can switch between them, in which case it is referred to as being a *switch branch*.

Both the potential and the flow of a source branch are accessible in expressions anywhere in the module. The models for potential and flow sources are shown below:



`f is a probe that measures the flow through the branch, and p is a probe that measures the potential across the branch.`

Figure 5-1: Equivalent circuit models for source branches.

5.1.2.2 Probes

If no value is specified for either the potential or the flow, the branch is a *probe*. If the flow of the branch is used in an expression anywhere in the module, the branch is a *flow probe*, otherwise the branch is a *potential probe*. Using both the potential and the flow of a probe branch is considered illegal. The models for probe branches are shown below

.

Figure 5-2: Equivalent circuit models for probe branches.

The branch potential of a flow probe is zero. The branch flow of a potential probe is zero.

5.1.3 Examples

The following examples demonstrate how to formulate models and the correspondence between the behavioral description and the equivalent probe/source model.

For simplification, only the node or branch declarations and contribution statements are shown.

5.1.3.1 The Four Controlled Sources

The model for a voltage controlled voltage source is.

```
branch (ps,ns) in, (p,n) out;
V(out) <+ A * V(in);
```

The model for a voltage controlled current source is.

```
branch (ps,ns) in, (p,n) out;
I(out) <+ A * V(in);
```

The model for a current controlled voltage source is.

```
branch (ps,ns) in, (p,n) out;
V(out) <+ A * I(in);
```

The model for a current controlled current source is.

```
branch (ps,ns) in, (p,n) out;
I(out) <+ A * I(in);
```

5.1.3.2 Resistor and Conductor

The model for a linear conductor is

```
branch (p,n) cond;
I(cond) <+ G * V(cond);
```

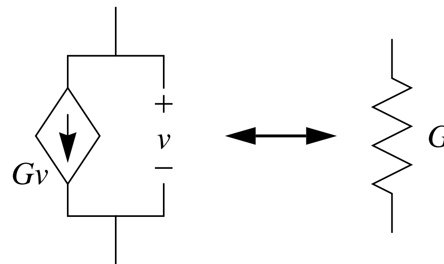


Figure 5-3: Linear conductor model

The assignment to `I(cond)` makes `cond` a current source branch and `V(cond)` simply accesses the optional potential probe built into the current source branch.

The model for a linear resistor is

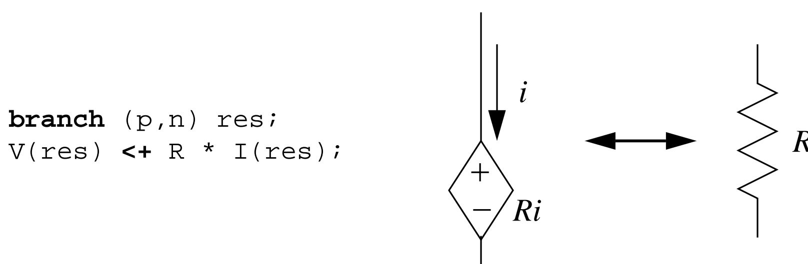


Figure 5-4: Linear resistor model

The assignment to `V(res)` makes `res` a potential source branch and `I(res)` simply accesses the optional flow probe built into the potential source branch.

5.1.3.3 RLC Circuits

A series RLC circuit is formulated by summing the voltage across the three components

.

$$v(t) = Ri(t) + L\frac{d}{dt}i(t) + \frac{1}{C}\int_{-\infty}^{t} i(\tau)d\tau$$

It is described as

```
V(p,n) <+ R*I(p,n) + L*ddt(I(p.n)) + idt(I(p,n))/C;
```

A parallel RLC circuit is formulated by summing the currents through the three components

.

$$i(t) = \frac{v(t)}{R} + C\frac{d}{dt}v(t) + \frac{1}{L}\int_{-\infty}^{t} v(\tau)d\tau$$

It is described as

```
I(p,n) <+ V(p,n)/R + C*ddt(V(p,n)) + idt(V(p,n))/L;
```

5.1.3.4 Simple Implicit Diode

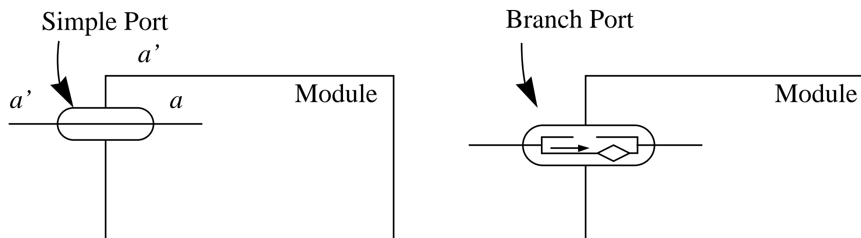
Verilog-A HDL allows components to be described with implicit equations. In the following example, which is a simple diode with a series resistor, the model is implicit because the diode current `I(a,c)` appears on both sides of the contribution operator. The current of the diode branch is specified, making it a flow source branch. In addition, both the voltage and current of diode branch is used in the behavioral description.

```
I(a,c) <+ is * ($limexp((V(a,c) - rs * I(a,c)) / Vt) - 1);
```

5.1.4 Port Branches

With the methods of accessing signals at ports already described, it is possible to access the signals on the node to which the port is connected. It is also possible to treat the port itself as a restricted branch such that the flow through the port branch may be probed. Promoting port branches to support all of the capabilities of branches is under consideration. Until then, it is not permitted to set the flow or access the potential of a port branch.

This is shown schematically as




```
With a simple port, both sides of the port are
indistinguishable. Using port branches the ports
implement a probe branch model.
```

Figure 5-5: Branch Port

In the following discussions, *a* represents the inside terminal of a branch port, and *a'* represents the outside terminal. The terminal *a'* can not be accessed from within the module definition and the branch port is denoted by simply giving the port name *a* twice.

To access the flow of a port, the flow access function is used with the port name appearing twice as the argument list. For example,

`I(a,a)` accesses the current through port branch *a*.

It is also possible to declare a named port branch and use it as a conventional named branch for flow access.

```
branch (a,a) in;
```

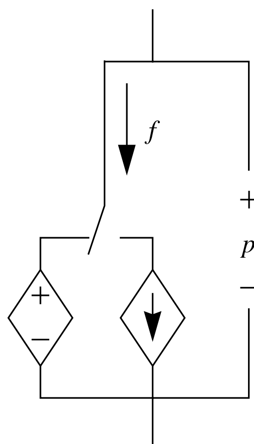
As one example of how this capability might be used, consider the *junction diode* re-written such that the total diode current is monitored and a message is issued if it exceeds a given value:

```
module diode (a, c);
electrical a, c;
branch (a, c) diode, cap, (a, a) anode;
parameter real is=1e-14, tf=0, cjo=0, imax=1, phi=0.7 ;

analog begin
   I(diode) <+ is*($limexp(V(diode)/$vt) - 1);
   I(cap) <+ ddt(tf*I(diode) - 2*cjo*sqrt(phi * (phi * V(cap))));

   if (I(anode) > imax)
      $strobe( "Warning: diode is melting!" );
end
endmodule
```

The expression V(a,a) is invalid for ports and nodes, where V is the potential access function.

5.1.5 Switch Branches

Source branches have the ability to switch between being potential and flow sources. To switch a branch to being a potential source, assign to its potential. To switch a branch to being a flow source, assign to its flow. This type of branch is useful when modeling ideal switches and mechanical stops. The full circuit model for a branch is shown below



Position of the switch depends on whether a potential or flow is assigned to the branch.

Figure 5-6: Circuit model for a source branch.

An ideal relay (a controlled switch) can be implemented as

```
if (closed)
   V(p,n) <+ 0;
```

```
else
   I(p,n) <+ 0;
```

A discontinuity of order zero is assumed to occur when the branch switches and so it is not necessary to use the *discontinuity* function with switch branches.

5.1.6 Unassigned Sources

If a value is not assigned to a branch, the branch flow is set to zero.

Consider

```
if (closed)
   V(p,n) <+ 0;
```

This example is equivalent to

```
if (closed)
   V(p,n) <+ 0;
else
   I(p,n) <+ 0;
```

5.2 Contribution statements

Verilog-A HDL defines the *branch contribution operator* "**<+**" for the description of analog behavior. This operator is only valid within the *analog block*. Branch contribution statements are statements that use the branch contribution operators to describe behavior in terms of a mathematical mapping of input signals to output signals.

5.2.1 Branch Contribution Statements

In general, a branch contribution statement consists of two parts, a left-hand side, and a right-hand side separated by a branch contribution operator. The right-hand side can be any expression that evaluates to a real value. The left-hand side specifies the source branch signal that the right-hand side is to be assigned to. It must consist of an access function applied to a branch. Hence, analog behaviors can be described using:

```
V(n1,n2) <+ expression ;
```

or

```
I(n1,n2) <+ expression ;
```

where `(n1, n2)` represents an unnamed source branch, and `V(n1,n2)` refers to the potential on the branch while `I(n1,n2)` refers to the flow through the branch. The expression can be linear, nonlinear, or dynamic in nature.

Branch contribution statements implicitly define source branch relations. The branch is directed from the first node of the access function to the second node. If the second node is not specified, ground is taken as the reference node.

A branch relation is a path of the flow between two nodes in a module. Each node has two signals associated with it—the potential of the node and the flow out of the node. In electrical circuits, the potential of a node is its voltage, whereas the flow out of the node is its current. Similarly, each branch has two signals associated with it—the potential across the branch and the flow through the branch.

For source branch contributions, the statement is evaluated as follows:

1. The simulator evaluates the right-hand side.
2. The simulator adds the value of the right-hand side to any previously retained value for the branch for later assignment to the branch. If there are no previously retained values, the value of the right-hand side itself is retained.
3. At the end of the simulation cycle, the simulator assigns the retained value to the source branch.

Contributing a flow to a branch that already has a value retained for the potential results in the potential being discarded and the branch being converted to a flow source. Conversely, contributing a potential to a branch that already has a value retained for the flow results in the flow being discarded and the branch being converted into a potential source.

The syntax for source contribution statement is shown below:

.

```
branch_contribution ::=
      bvalue <+ expression ;

bvalue ::=
      access_identifier ( analog_signal_list )

analog_signal_list ::=
      branch_identifier
    | node_or_port_identifier
    | node_or_port_identifier , node_or_port_identifier
```

Figure 5-7: Syntax for branch contribution

5.2.2 Indirect Branch Assignments

Verilog-A HDL allows descriptions that implicitly specify a branch voltage or current in fixed-point form. The branch voltage or current is assigned a value by an expression that uses the branch voltage or current. This occurred in the simple implicit diode model above where `I(a,c)` appeared on both sides of the contribution operator.

Consider the model for an ideal OPAMP. In this model, the output is driven to the voltage that results in the input voltage being zero. The constitutive equation is

```
V(in) == 0
```

This can be formulated in fixed point form as

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

This statement defines the output of the OPAMP to be a controlled voltage source by assigning to `V(out)` and defines the input to be high impedance by only probing the input voltage. The desired behavior results because the description is formulated in such a way that it reduces to `V(in) = 0`. This approach does not result in the right tolerances being applied to the equation if `out` and `in` have different disciplines.

Verilog-A HDL includes a special syntax that is appropriate in this situation. The above branch contribution can be rewritten using an *indirect branch assignment*:

```
V(out): V(in) == 0;
```

which reads "find `V(out)` such that `V(in) == 0`". It indicates that `out` should be driven with a voltage source and the source voltage should be such that the given equation is satisfied. Any branches referenced in the equation are only probed and not driven. In particular, `V(in)` acts as a voltage probe.

The syntax for the indirect assignment statement is

```
indirect_branch_assignement ::=
      target : equation ;

target ::=
      bvalue

equation ::=
      nexpr == expression

nexpr ::=
      bvalue
    | ddt ( bvalue )
    | idt ( bvalue )
```

Figure 5-8: Syntax for indirect branch assignment

If there are multiple indirect assignments statements, it is often the case that the targets can be paired with any equation. Consider the following ordinary differential equation,

$$\frac{dx}{dt} = f(x, y, z)$$

$$\frac{dy}{dt} = g(x, y, z)$$

$$\frac{dz}{dt} = h(x, y, z)$$

which can be written as

```
V(x): ddt(V(x)) == f(V(x), V(y), V(z));
V(y): ddt(V(y)) == g(V(x), V(y), V(z));
V(z): ddt(V(z)) == h(V(x), V(y), V(z));
```

or

```
V(y): ddt(V(x)) == f(V(x), V(y), V(z));
V(z): ddt(V(y)) == g(V(x), V(y), V(z));
V(x): ddt(V(z)) == h(V(x), V(y), V(z));
```

or

```
V(z): ddt(V(x)) == f(V(x), V(y), V(z));
V(x): ddt(V(y)) == g(V(x), V(y), V(z));
V(y): ddt(V(z)) == h(V(x), V(y), V(z));
```

without affecting the results.

5.2.2.1 Indirect Assignment and Contribution

Indirect assignment is incompatible with contribution. Once a value is indirectly assigned to a branch, it cannot be contributed to using the branch contribution operator (‘**<+**’).

Section 6

Analog Behavior

The description of an analog behavior consists of setting up contributions (Section 5) for various nodes under certain procedural or timing control. This section describes an analog procedural block, procedural control statements and analog timing control functions.

6.1 Analog procedural block

Discrete behavioral definitions within Verilog HDL are encapsulated within the *initial* and *always* procedural blocks. Every *initial* and *always* block starts a separate concurrent activity flow. For continuous time simulation, the behavioral description is encapsulated within the *analog* procedural block. Verilog-A HDL supports one analog procedural block in a module definition.

The *analog* procedural block defines the behavior as a procedural sequence of statements. The conditional and looping constructs are available for defining behaviors within the *analog* procedural block. Because the description is a continuous-time behavioral description, no blocking event control statements (such as blocking delays, events or waits) are supported.

The syntax for analog block is as follows:

```
analog_block ::=
   analog statement
statement ::=
     null_statement
   | block_statement
   | branch_contribution
   | indirect_branch_assignment
   | procedural_assignment
   | conditional_statement
   | looping_statement
   | case_statement
   | generate_statement
   | event_controlled_statement
   | discontinuity_function
   | bound_step_function
   | last_crossing_function
   | system_task_enable
```

Figure 6-1: Syntax for analog procedural block

The statements within the analog block are used to define the large-signal response of the module. The behavioral description is a mathematical mapping of input signals to output signals. The mapping is done with contribution statements of the form

```
signal <+ expression ;
```

The *expression* can be any combination of linear, nonlinear, or differential expressions of module signals, constants and parameters (see Section 5).

The analog block is executed continuously. The simulator calculates the time advance by computing the time step based on convergence. The procedural statements in the analog block are executed sequentially. The contributions help form the differential equations to compute flow and potential values at various nodes.

All analog blocks contained in various modules in a design are considered to be executing concurrent with respect to each other.

6.2 Null statement

A *null statement* is a statement that does nothing. It is represented by a semicolon. The analog procedural block is not allowed to have only a null statement. That is, the following analog procedural block is illegal

```
analog ;
```

A null statement in presence of at least one other statement is allowed.

The syntax for null statement is as follows:

```
null_statement ::=
   ;
```

Figure 6-2: Syntax for Null Statement

6.3 Block statement

The *block statement*, also referred to as *sequential block*, is a means of grouping two or more statements together so that they act syntactically like a single statement. The block statement is delimited by the keywords **begin** and **end**. The procedural statements in a block statement are executed sequentially in the given order.

The following is the formal syntax for a sequential block:

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

block_item_declaration ::=
     parameter_declaration
   | integer_declaration
   | real_declaration
```

Figure 6-3: : Syntax for the sequential block

6.3.1 Block names

A sequential block can be named by adding `: name_of_block` after the keyword **begin**. The naming of a block allows local variables to be declared for the block.

All local variables are static—that is, a unique location exists for all variables and leaving or entering blocks do not affect the values stored in them.

The block names give a means of uniquely identifying all variables at any simulation time.

6.4 Procedural assignment

In Verilog-A HDL, the branch contributions and indirect branch assignments are used for modifying signals. The procedural assignments are used for modifying integer and real variables. The syntax for procedural assignment is as follows:

```
procedural_assignment ::=
     lexpr = expression ;

lexpr ::=
     integer_identifier
   | real_identifier
   | array_element

array_element ::=
     integer_identifier [ constant_expression ]
   | real_identifier [ constant_expression ]
```

Figure 6-4: Syntax for procedural assignment

The left-hand side of a procedural assignment must be an integer or a real identifier or an element of an integer or real array. The right-hand side expression can be any arbitrary expression constituted from legal operands and operators as described in Section 4.

6.5 Conditional statement

The *conditional statement* (or *if-else* statement) is used to make a decision as to whether a statement is executed or not. The syntax of a conditional statement is as follows:

```
conditional_statement ::=
     if ( expression ) statement
     [ else statement ]
```

Figure 6-5: : Syntax of conditional statement

If the expression evaluates to true (that is, has a non-zero value), the first statement will be executed. If it evaluates to false (has a zero value), the first statement will not be executed. If there is an *else statement* and expression is false, the else statement will be executed.

Since the numeric value of the `if` expression is tested for being zero, certain shortcuts are possible. For example, the following two statements express the same logic:

```
if (expression)
if (expression != 0)
```

Because the else part of an if-else is optional, there can be confusion when an else is omitted from a nested if sequence. This is resolved by always associating the else with the closest previous if that lacks an else. In the example below, the else goes with the inner if, as shown by indentation.

```
if (index > 0)
   if (i > j)
      result = i;
   else       // else applies to preceding if
      result = j;
```

If that association is not desired, a *begin-end block statement* must be used to force the proper association, as shown below.

```
if (index > 0) begin
   if (i > j)
      result = i;
end
else result = j;
```

6.5.1 If-else-if Construct

The following construction occurs so often that it is worth a brief separate discussion:

```
if_else_if_statement ::=
   if (expression) statement
   { else if (expression) statement }
   else statement
```

Figure 6-6: : Syntax of if-else-if construct

This sequence of if statements (known as an *if-else-if* construct) is the most general way of writing a multi-way decision. The expressions are evaluated in order; if any expression is true, the statement associated with it will be executed, and this will terminate the whole chain. Each statement is either a single statement or a sequential block of statements.

The last else part of the if-else-if construct handles the none-of-the-above or default case where none of the other conditions were satisfied. Sometimes there is no explicit action for the default; in that case, the trailing else statement can be omitted or it can be used for error checking to catch an impossible condition.

6.6 Case statement

The *case statement* is a multi-way decision statement that tests whether an expression matches one of a number of other expressions, and branches accordingly. The case statement has the following syntax:

```
case_statement ::=
     case   ( expression ) case_item { case_item } endcase

case_item ::=
     expression { , expression } : statement
   | default [ : ] statement
```

Figure 6-7: : Syntax for case statement

The *default* statement is optional. Use of multiple default statements in one case statement is illegal.

The case expression and the case item expression can be computed at runtime; neither expression is required to be a constant expression.

The *case item expressions* are evaluated and compared in the exact order in which they are given. During the linear search, if one of the `case` item expressions matches the case expression given in parentheses, then the statement associated with that case item is executed. If all comparisons fail, and the default item is given, then the default item statement is executed. If the default statement is not given, and all of the comparisons fail, then none of the case item statements are executed.

6.6.1 Constant expression in case statement

A constant expression can be used for case expression. The value of the constant expression shall be compared against case item expressions.

The following example demonstrates the usage by modeling a 3-bit priority encoder.

```
integer [2:0] encode ;

case (1)
   encode[2] : $display("Select Line 2") ;
   encode[1] : $display("Select Line 1") ;
   encode[0] : $display("Select Line 0") ;
   default $strobe("Error: One of the bits expected ON");
endcase
```

Note that the case expression is a constant expression (1). The case items are expressions (array elements), and are compared against the constant expression for a match.

6.7 Looping statements

There are four kinds of looping statements. These statements provide a means of controlling the execution of a statement zero, one, or more times.

forever repeatedly executes a statement.

repeat executes a statement a fixed number of times. Evaluation of the constant expression decides how many times a statement is executed.

while executes a statement until an expression becomes false. If the expression starts out false, the statement is not executed at all.

for controls execution of its associated statement(s) by a three-step process, as follows:

1. executes an assignment normally used to initialize an integer that controls the number of loops executed
2. evaluates an expression—if the result is zero, the for-loop exits, and if it is not zero, the for-loop executes its associated statement(s) and then perform step 3.
3. executes an assignment normally used to modify the value of the loop-control variable, then repeats step 2 above.

The following shows the syntax for various looping statements:

```
looping_statement ::=
     forever statement
   | repeat ( expression ) statement
   | while ( expression ) statement
   | for ( procedural_assignment ; expression ;
           procedural_assignment ) statement
```

Figure 6-8: : Syntax for the looping statements

Analog operators are not allowed in any of the four looping statements.

6.8 Generate statement

The *generate statement* is a looping construct that is unrolled at elaboration time. Generate statement is the only looping statement that can contain analog operators.

The syntax of generate statement is as follows:

.

```
generate_statement ::=
   generate index_identifier ( start_expr, end_expr [, incr_expr ] )
            statement

start_expr ::=
   constant_expression

end_expr ::=
   constant_expression

incr_expr ::=
   constant_expression
```

Figure 6-9: Syntax for generate statement

The index must not be assigned or modified in any way inside the loop. In addition, it is local to the loop and is expanded when the loop is unrolled. Even if there is a local variable with the same name as the index and the variable is modified as a side effect of a function called from within the loop, the loop index is unaffected.

The start and end bounds and increment are constant expressions. They are only evaluated at elaboration time. If the expressions used for the increment and bounds change during the simulation, it does not affect the behavior of the generate statement.

If the lower bound is less than the upper bound and the increment is negative, or if the lower bound is greater than the upper bound and the increment is positive, then the generate statement does not execute.

If the lower bound equals the upper bound, the increment is ignored and the statement execute once. If the increment is not given, it is taken to be +1 if the lower bound is less than the upper bound, and -1 if the lower bound is greater than the upper bound.

The statement, which can be a sequential block, is replicated with all occurrences of index in the statement replaced by a constant. In the first instance of the statement, the index is replaced with the lower bound. In the second, it is replaced by the lower bound plus the increment. In the third, it is replaced by the lower bound plus two times the increment. This pattern is repeated until the lower bound plus a multiple of the increment is greater than the upper bound.

Example:

This module implements a continuously running (not clocked) analog-to-digital converter.

```
module adc(in,out) ;
   parameter bits=8, fullscale=1.0, delay=0.0, ttime=10n ;
   input in ;
   output [0:bits-1] out ;
   electrical in, out ;
   real sample, thresh ;

   analog begin
      thresh = fullscale/2.0 ;
      generate i (bits-1,0) begin
           V(out[i]) <+ transition(sample > thresh, delay, ttime) ;
           if (sample > thresh) sample = sample - thresh ;
           sample = 2.0 * sample ;
      end
   end
endmodule
```

6.9 Analog events

The analog behavior of a component can be controlled using analog events. The analog events have the following characteristics:

1. analog events have no time duration
2. analog events can be triggered and detected in different parts of the behavioral model.
3. analog events do not block the execution of an analog block
4. analog events can be detected using `@` *operator*

5. analog events do not hold any data

There are two types of analog events - *global events* and *monitored events*.

6.9.1 Event detection

Analog event detection consist of an event expression followed by a procedural statement. It takes the form:

```
event_controlled_statement ::=
     @ ( event_expression ) statement

event_expression ::=
     simple_event [ or event_expression ]

simple_event ::=
     global_event
   | event_function
   | identifier
```

Figure 6-10: Syntax for event detection

The procedural statement following the event expression is executed whenever the event expression changes. The analog event detection is non-blocking, meaning that the execution of the procedural statement is skipped unless the analog event has occurred. The event expression consists of one or more signal names, global events, or monitored events separated by `or` operator.

The parenthesis around the event expression are required.

6.9.2 Event OR operator

The "OR-ing" of any number of events can be expressed such that the occurrence of any one of the events trigger the execution of the procedural statement that follows it. The keyword `or` is used as an event or operator.

For example,

```
analog begin
   @(initial_step or cross(V(smpl)-2.5,+1))
      V(out) <+ 0 ;
end
```

Here, `initial_step` is a global event and `cross()` returns a monitored event. `V(out)` is set to `0` when one of the two events occur.

6.9.3 Global events

The global events are generated by the simulator at various stages of the simulation. The user model can not generate these events. These events are detected by using the name of the global event in an event expression with the @ operator.

The global events are pre-defined in Verilog-A HDL. These events can not be redefined in a model.

The following are pre-defined global events:

```
global_event ::=
     initial_step [ ( analysis_list ) ]
   | final_step [ ( analysis_list ) ]

analysis_list ::=
     analysis_name { , analysis_name }

analysis_name ::=
     " analysis_identifier "
```

Figure 6-11: Global events

The *initial_step* and *final_step* generate global events on the first and the last time-step in a simulation respectively. They are useful when performing actions that should only occur at the beginning or end of an analysis.

The following example measures the bit-error rate of a signal and prints the result at the end of the simulation.

```
module bitErrorRate (in, ref) ;
   input in, ref ;
   electrical in, ref ;
   parameter real period=1, thresh=0.5 ;
   integer bits, errors ;

   analog begin
      @(initial_step) begin
           bits = 0 ;
           errors = 0 ;
      end

      @(timer(0, period)) begin
           if ((V(in) > thresh) != (V(ref) > thresh))
              errors = errors + 1 ;
           bits = bits + 1 ;
      end

      @(final_step)
        $strobe("bit error rate = %f%%", 100.0 * errors / bits ) ;
   end
endmodule
```

The initial_step and final_step events take a list of quoted strings as optional arguments. The strings are compared to the name of the analysis being run. If any string matches the name of the current analysis name, then the simulator generates an event on the first step and the last step of that particular analysis, respectively.

If no arguments are present, then the simulator generates event in any analysis where time is the independent variable (such as a transient analysis).

6.9.4 Monitored events

The monitored events are detected using event functions with the @ operator. The triggering of the monitored event is implicit due to change in signals, simulation time, or other runtime conditions.

```
event_function ::=
     cross_function
   | timer_function
```

Figure 6-12: Monitored events

6.9.4.1 Cross Function

The *cross function* is used for generating a monitored analog event to detect threshold crossings in analog signals.

```
cross_function ::=
   cross ( expression [ , opt_args ] )

opt_args ::=
   direction [ , time_tol [ , expression_tol ] ]

direction ::=
   +1 | -1

time_tol ::=
   expression

expression_tol ::=
   expression
```

Figure 6-13: Syntax for Cross function

The cross function generates events when the expression crosses zero in the specified direction. In addition, cross controls the timestep to accurately resolve the crossing. If the direction indicator is 0 or not specified, then the event and timestep control occur on both positive and negative crossings of the signal. If direction indicator is +1 (–1), then

the event and timestep control only occurs on positive (negative) transitions of the signal. For any other transitions of the signal, the cross function does not generate an event.

Both the time and expression tolerances must be positive. If the expression tolerance is required, both the time and expression tolerances must be satisfied at the crossing.

The following description of a sample-and-hold illustrates how the *cross* function might be used.

```
module sh (in, out, smpl) ;
    output out ;
    input in, smpl ;
    electrical in, out, smpl ;
    real state ;

    analog begin
        @(cross(V(smpl) - 2.5, +1))
             state = V(in) ;
        V(out) <+ transition(state, 0, 10n) ;
    end
endmodule
```

The cross function maintains internal state and has the same restrictions as analog operators. In particular, it must not be used inside a conditional statement (if and case) unless the conditional expression that controls the statement consists of terms that cannot change its value during the course of an analysis. In particular, the conditional expression can only consist of constants, parameter values, and the analysis() function. In addition, cross function is not allowed in the forever, repeat, while, and for iteration statements. It is allowed in generate statement.

6.9.4.2 Timer Function

The *timer function* is used to generate analog event to detect specific points in time.

```
timer_function ::=
     timer ( start_time [ , period ] )

start_time ::=
     expression

period ::=
     expression
```

Figure 6-14: Syntax for timer function

The timer function schedules an event that occurs at an absolute time (as specified by *start_time*). The analog simulator places a time point at, or just beyond, the time of the event. If *period* is specified, then the timer function schedules subsequent events at multiples of the period.

A pseudo-random bit stream generator is an example how the timer function might be used.

```
module bitStream (out) ;
   output out ;
   electrical out ;
   parameter period = 1.0 ;
   integer x ;

   analog begin
      @(timer(0, period))
           x = $random + 0.5 ;
      V(out) <+ transition( x, 0.0, period/100.0 ) ;
   end
endmodule
```

6.10 Announcing Discontinuity

The *discontinuity* function is used to give hints to the simulator about the behavior of the module so that it can control the simulation algorithms to get accurate results in exceptional situations. It does not directly specify the behavior of the module. The discontinuity function should be executed whenever the analog behavior changes discontinuously.

Because discontinuous behavior can cause convergence problems, discontinuity should be avoided whenever possible.

The filter functions (transition, slew, laplace, etc.) are provided to smooth discontinuous behavior. However, in some cases it is not possible to implement the desired functionality using these filters. In this case, *discontinuity* function should be executed when the signal behavior changes abruptly.

```
discontinuity_function ::=
     discontinuity ( constant_expression )
```

Figure 6-15: Syntax for discontinuity function

The *discontinuity* function takes one integer argument that indicates the degree of the discontinuity. Discontinuity(i) would imply that there is a discontinuity in the *i*'th derivative of the constitutive equation with respect to either a signal value or time where *i* must be non-negative. Hence, discontinuity(0) indicates a discontinuity in the equation, discontinuity(1) indicates a discontinuity in its slope, etc.

Discontinuity created by switch branches and built-in system functions, such as transition() and slew() do not need to be announced.

The following example uses the discontinuity function to model a relay.

```
module relay (c1, c2, pin, nin) ;
   inout c1, c2 ;
   input pin, nin ;
   electrical c1, c2, pin, nin ;
   parameter real r=1 ;

   analog begin
      @(cross(V(pin,nin))) discontinuity(0) ;
      if (V(pin,nin) >= 0)
           I(c1,c2) <+ V(c1,c2)/r;
      else
           I(c1,c2) <+ 0 ;
   end
endmodule
```

In this example, cross function controls the time step so that the time when the relay changes position is accurately resolved. It also triggers the discontinuity function that causes the simulator to react properly to the discontinuity. This would have been handled automatically if the type of the branch (c1,c2) had been switched between voltage and current.

Another example is a source that generates a triangular wave. In this case, neither the model nor the waveforms generated by the model are discontinuous. Rather, the waveform generated is piecewise linear with discontinuous slope. If the simulator is aware of the abrupt change in slope, it can adapt the integration method to eliminate problems that result from the discontinuous slope (typically changing to a first order integration method).

```
module triangle (out) ;
   output out ;
   voltage out ;
   parameter real period = 10.0, amplitude = 1.0 ;
   integer slope ;
   real offset ;

   analog begin
      @(timer(0, period) begin
           slope = +1 ;
           offset = $realtime ;
           discontinuity(1) ;
      end
      @(timer(period/2, period) begin
           slope = -1 ;
           offset = $realtime ;
           discontinuity(1) ;
      end
      V(out) <+ amplitude * slope *
                         (4 * ($realtime - offset) / period - 1) ;
   end
endmodule
```

Finally, here is a case where timer function is used without using a discontinuity function. In this case, the event generated by the timer() function indicates that a measurement should be printed, but that neither the model nor the waveforms contain discontinuity. In this case, switching to a first order integration method would result in a degradation of accuracy.

```
module sampler (in) ;
   input in ;
   voltage in ;
   parameter real period = 10.0 ;

   analog @(timer(0, period))
      $strobe("%g\t%g", $realtime, V(in)) ;
endmodule
```

6.11 Time related functions

There are two functions, bound_step and last_crossing, related to simulation time.

6.11.1 Bounding the time step

The *bound_step function* puts a bound on the next time step. It does not specify exactly what the next time step should be, but it bounds how far the next time point can be from the present time point. The function takes the maximum time step as an argument. It does not return a value. The syntax is as follows:

```
bound_step_function ::=
     bound_step ( max_step )

max_step ::=
     constant_expression
```

Figure 6-16: Syntax for bound_step function

The example below implements a sinusoidal voltage source and uses the bound_step() function to assure that the simulator faithfully follows the output signal (it is forcing 20 points per cycle).

```
module vsine(out) ;
   output out ;
   voltage out ;
   parameter real freq=1.0, ampl=1.0, offset=0.0 ;
```

```
    analog begin
        V(out) <+ ampl * sin(2.0 * `M_PI * freq * $realtime) + offset ;
        bound_step(0.05 / freq) ;
    end
endmodule
```

6.11.2 Last_Crossing Function

Related to the cross function, the *last_crossing function* returns the simulation time when a signal expression last crossed 0.

```
last_crossing_function ::=
     last_crossing ( expression [ , direction ] )
```

Figure 6-17: Syntax for last_crossing function

The *direction* flag is interpreted in the same way as with the cross function. The last_crossing function is subject to the same usage restrictions as the cross function.

The last_crossing function does not control the timestep to get accurate results, and uses linear interpolation to estimate the time of the last crossing. However, it can be used with the cross function for improved accuracy.

The following example measures the period of its input signal using cross and last_crossing functions.

```
module period(in) ;
    input in ;
    voltage in ;
    integer crossings ;
    real latest, previous ;

    analog begin
        @(initial_step) begin
             crossings = 0 ;
             previous = 0 ;
        end

        @(cross(V(in), +1)) begin
             crossings = crossings + 1 ;
             previous = latest ;
        end
        latest = last_crossing(V(in), +1) ;
```

```
            @(final_step) begin
                 if (crossings < 2)
                    $strobe("Could not measure period.") ;
                 else
                    $strobe("period = %g, crossings = %d",
                                               latest-previous, crossings) ;
            end
         end
      endmodule
```

Before the expression crosses zero for the first time, the last_crossing function returns a **-inf**.

Section 7

Hierarchical Structures

Verilog-A HDL supports a hierarchical hardware description by allowing modules to be embedded within other modules. Higher-level modules create instances of lower-level modules and communicate with them through input, output, and bidirectional ports.

To describe a hierarchy of modules, the user provides textual definitions of various modules. Each module definition stands alone; the definitions are not nested. Statements within the module definitions create instances of other modules, thus describing the hierarchy.

7.1 Modules

A module definition is enclosed between the keywords **module** and **endmodule**. The identifier following the keyword **module** is the name of the module being defined. The optional list of ports specify an ordered list of the module's ports. The order used can be significant when instantiating the module (section 7.1.2). The identifiers in this list must be declared in input, output, and inout declaration statements within the module definition. The module items define what constitutes a module, and include many different types of declarations and definitions. A module definition can have at most one analog block.

```
module_declaration ::=
     module module_identifier [ list_of_ports ] ;
     [ module_items ]
     endmodule

list_of_ports ::=
   ( port { , port } )

port ::=
     port_expression
   | . port_identifier ( [ port_expression ] )

port_expression ::=
     port_identifier
   | port_identifier [ constant_expression ]
   | port_identifier [ constant_range ]

constant_range ::=
     msb_constant_expression : lsb_constant_expression

module_items ::=
     { module_item }
   | analog_block

module_item ::=
     module_item_declaration
   | parameter_override
   | module_instantiation

module_item_declaration ::=
     parameter_declaration
   | input_declaration
   | output_declaration
   | inout_declaration
   | integer_declaration
   | node_declaration
   | real_declaration

parameter_override ::=
     defparam list_of_param_assignments ;
```

Figure 7-1: Syntax for module

7.1.1 Top-level modules

Top-level modules are modules that are included in the source text but are not instantiated, as described in section 7.1.2.

7.1.2 Module instantiation

Instantiation allows one module to incorporate a copy of another module into itself. Module definitions do not nest. That is, one module definition does not contain the text of another module definition within its **module-endmodule** keyword pair. A module definition nests another module by *instantiating* it. The *module instantiation statement* creates one or more named *instances* of a defined module.

The following is the syntax for specifying instantiations of modules:

```
module_instantiation ::=
     module_identifier [ parameter_value_assignment ] instance_list

parameter_value_assignment ::=
     # ( ordered_param_override_list )
   | # ( named_param_override_list )

ordered_param_override_list ::=
     expression { , expression }

named_param_override_list ::=
     named_param_override { , named_param_override }

ordered_param_override ::=
   . parameter_identifier ( expression )

instance_list ::=
     module_instance { , module_instance } ;

module_instance ::=
     name_of_instance  ( [ list_of_module_connections ] )

name_of_instance ::=
     module_instance_identifier [ range ]

list_of_module_connections ::=
     ordered_port_connection { , ordered_port_connection }
   | named_port_connection { , named_port_connection }

ordered_port_connection ::=
     [ expression ]

named_port_connection ::=
     . port_identifier ( [ expression ] )

range ::=
     [ constant_expression : constant_expression ]
```

Figure 7-2: : Syntax for module instantiation

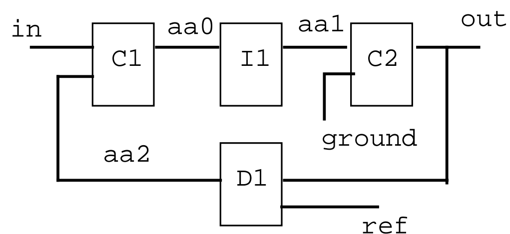
The instantiations of modules can contain a range specification. This allows an array of instances to be created.

One or more module instances (identical copies of a module definition) can be specified in a single module instantiation statement.

The list of module connections can be provided only for modules defined with ports. The parentheses, however, are always required. When a list of module connections is given, the first element in the list connects to the first port, the second to the second port, and so on. See section 7.3 for a more detailed discussion of ports and port connection rules.

A connection can be a simple reference to a node identifier or a sub-range of a vector node. The example below illustrates a comparator and an integrator (lower-level modules) which are instantiated in sigma-delta A/D converter module (the higher-level module).

```
module comparator(cout, inp, inm);
output cout;
input inp, inm;
electrical cout, inp, inm;
parameter real td = 1n, tr = 1n, tf = 1n;

analog begin
   @cross(V(inp) - V(inm), 0)
      V(cout) <+ transition((V(inp) > V(inm)) ? 1 : 0, td, tr, tf);
end
endmodule

module integrator(out, in);
output out;
input in;
electrical in, out;
parameter real gain = 1.0;
parameter real ic = 0.0;

analog begin
   V(out) <+ gain*idt(V(in), ic);
end
endmodule
```




```
module sigmadelta(out, ref, in);
output out;
input ref, in;

comparator C1(.cout(aa0), .inp(in), .inm(aa2));
integrator #(1.0) I1(.out(aa1), .in(aa0));
comparator C2(out, aa1, ground);
d2a #(.width(1)) D1(aa2, ref, out);   // A D/A converter

endmodule
```

The comparator instance C1 and the integrator instance I1 use named port connections, whereas the comparator instance C2 and the d2a (not described here) instance D1 uses ordered port connection.

The integrator instance I1 overrides gain parameter positionally, whereas the d2a instance D1 overrides width parameter by named association.

7.2 Overriding module parameter values

When one module instantiates another module, it can alter the values of any parameters declared within the instantiated module. There are three ways to alter parameter values: the *defparam statement*, which allows assignment to parameters using their hierarchical names, *module instance parameter value assignment by order*, which allows values to be assigned in-line during module instantiation in the order of their declaration, and *module instance parameter value assignment by name*, which allows values to be assigned in-line during module instantiation by explicitly associating parameter names with the overriding values.

7.2.1 Defparam statement

Using the *defparam statement*, parameter values can be changed in any module instance throughout the design using the hierarchical name of the parameter. See section 7.4 for hierarchical names.

The expression on the right hand side of the defparam assignments must be a constant expression involving only constant numbers and references to parameters. The referenced parameters (on the right hand side of the defparam) must be declared in the same module as the defparam statement.

The defparam statement is particularly useful for grouping all of the parameter value override assignments together in one module.

```
module tgate;
electrical io1,io2,control,control_bar;
mosn m1 (io1, io2, control);
mosp m2 (io1, io2, control_bar);
endmodule

module mosp (source,drain,gate);
   parameter gate_length= 0.3e-6,
             gate_width = 4.0e-6;

   spice_pmos #(.L(gate_length),.W(gate_width)) p(gate,source,drain);

endmodule

module mosn (source,drain,gate);
   parameter gate_length = 0.3e-6,
             gate_width = 4.0e-6;

   spice_nmos #(.L(gate_length),.W(gate_width)) n(gate,source,drain);

endmodule

module annotate;
defparam
      tgate.m1.gate_width = 5e-6,
      tgate.m2.gate_width = 10e-6;
endmodule
```

7.2.2 Module instance parameter value assignment by order

An alternative method for assigning values to parameters within module instances supplies values for particular instances of a module to any parameters that have been specified in the definition of that module.

The order of the assignments in module instance parameter value assignment must follow the order of declaration of the parameters within the module. It is not necessary to assign values to all of the parameters within a module when using this method. However, it is not possible to skip over a parameter assignment. Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters that make up this subset must precede the declarations of the remaining (optional) parameters. An alternative is to assign values to all of the parameters, but use the default value (the same value assigned in the declaration of the parameter within the module definition) for those parameters that do not need new values.

Consider the following example, where the parameters within module instance `mod_a` are changed during instantiation.

```
module m;
voltage clk;
electrical out_a, in_a;
electrical out_b, in_b;

// create an instance and set parameters
mosp #(2e-6,1e-6) weakp(out_a, in_a, clk);
// create an instance leaving default values
mosp plainp(out_b, in_b, clk);
endmodule
```

7.2.3 Module instance parameter value assignment by name

The third method of overriding parameters for a module instance is an explicit association between the name of the parameter and the new value being assigned to that parameter. The name of the parameter must be preceded by a period (.) and must be the name of a parameter in the definition of the module being instantiated. The overriding value for each parameter must be a constant expression and must be enclosed in parenthesis (`()`). Only those parameters whole value is being overridden need specification.

In the following example of instantiating a voltage-controlled oscillator, the parameters are specified on a named-association basis much they are for ports.

```
vco #(.centerFreq(5000), .convGain(1000)) vco1(lo_out, rf_in);
```

Here, the name of the instantiated vco module is *vco1*. The *centerFreq* parameter is passed a value of 5000, and the *convGain* parameter is passed a value of 1000. The positional assignment mechanism for ports assigns *lo_out* as the first node, and *rf_in* as the second node of *vco1*.

7.2.4 Parameter override precedence

If the value of a parameter is overridden using defparam statement as well as module instance parameter value assignments (see section 7.2.2 and section 7.2.3), the value assignment specified by the defparam statement is retained and the other value assignments are ignored.

If the value of a parameter is overridden using one of the three forms at different levels of module hierarchy, the value assignment done in the hierarchically highest level of module is retained and the other value assignments are ignored.

If the hierarchical relationship between the modules containing defparam statements cannot be determined, it must be reported as an error.

7.2.5 Parameter dependence

A parameter (for example, `gate_cap`) can be defined with an expression containing another parameter (for example, `gate_width` or `gate_length`). Since `gate_cap` depends on the value of `gate_width` and `gate_length`, a modification of `gate_width` or `gate_length` changes the value of `gate_cap`. For example, in the following parameter declaration, an update of `gate_width` , whether by defparam statement or in an instantiation statement for the module that defined these parameters, automatically updates `gate_cap`.

```
parameter
    gate_width = 0.3e-6,
    gate_length = 4.0e-6,
    gate_cap = gate_length * gate_width * `COX;
```

7.3 Ports

Ports provide a means of interconnecting instances of modules. For example, if a module A instantiates module B, the ports of module B are associated with either the ports or the internal nodes of module A. The top-level module does not have ports, so every port is eventually associated with a node.

7.3.1 Port association

The syntax for a port association is given below. It is the completion of the syntax presented in section 7.1.

```
port ::=
     port_expression
   | . port_identifier ( port_expression )
port_expression ::=
     port_identifier
   | port_identifier [ constant_expression ]
   | port_identifier [ constant_range ]
constant_range ::=
   msb_constant_expression : lsb_constant_expression
```

Figure 7-3: Syntax for port

The port expression in the port definition can be one of the following:

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

The two types of module port definitions cannot be mixed; the ports of a particular module definition must all be defined by order or all by name. The port expression is optional because ports can be defined that do not connect to anything internal to the module.

7.3.2 Port declarations

The type and direction of each port listed in the module definition's list of ports are declared in the body of the module.

7.3.2.1 Port type

The type of a port is declared by giving its discipline. If the type of a port is not declared, the port can only be used in a structural description (it can be passed to instances of modules, but cannot be accessed in a behavioral description).

```
node_declaration ::=
    discipline_identifier [ range ] port_identifiers;
port_identifiers ::=
    port_identifier { , port_identifier }
```

Figure 7-4: Syntax for port type declarations

7.3.2.2 Port direction

The direction of a port can be specified as **input**, **output**, or **inout** (bidirectional). If the direction is specified as being an input port, then the module will only monitor the signals at the port, and not modify them. That is, within the module the port can only be passed into other modules as input ports and the signals on the ports can only be used in expressions, they cannot be used on the left side of a contribution statement. If the direction is specified as being an output port, then the module will only affect the signals at the port, but not be affected by them. Thus, the port can be passed to instances of other modules as output ports and the signals on the ports cannot be used in expressions but can be used on the left side of a contribution statement. Finally, ports that are declared as being bidirectional are not subject to these restrictions. If the direction of the port is not specified, it is taken to be bidirectional. The syntax for port declarations is as follows:

```
input_declaration ::= input [ range ]  list_of_port_identifiers ;
output_declaration ::= output [ range ]  list_of_port_identifiers ;
inout_declaration ::= inout [ range ] list_of_port_identifiers ;
```

Figure 7-5: Syntax for port direction declarations

A port can be declared in both a port type declaration and a port direction declaration. If a port is declared as a vector, the range specification between the two declarations of a port must be identical.

Note: Implementations may limit maximum number of ports in a module definition, but will at least be 256.

7.3.3 Connecting module ports by ordered list

One method of making the connection between the ports listed in a module instantiation and the ports defined by the instantiated module is the ordered list—that is, the ports listed for the module instance must be in the same order as the ports listed in the module definition.

```
module adc4 (out, rem, in);
output [3:0] out ;   output rem;
input in;
electrical [3:0] out;
electrical in, rem, rem_chain;

adc2 hi2 (out[3:2], rem_chain, in) ;
adc2 lo2 (out[1:0], rem, rem_chain) ;
endmodule

module adc2 (out, remainder, in);
output [1:0] out ;   output remainder;
input in;
electrical [1:0] out ;
electrical in, remainder, r;

adc hi1 (out[1], r, in) ;
adc lo1 (out[0], remainder, r) ;
endmodule

module adc (out, remainder, in);
output out, remainder;
input in;
electrical out, in, remainder;
integer d;

   analog begin
      d = (V(in) > 0.5) ;
      V(out) <+ transition(d) ;
      V(remainder) <+ 2.0 * V(in) ;
      if (d)
         V(remainder) <+ -1.0 ;
   end
endmodule
```

7.3.4 Connecting module ports by name

The second way to connect module ports consists of explicitly linking the two names for each side of the connection—the name used in the module definition, followed by the name used in the instantiating module. This compound name is then placed in the list of module connections. The name of port must be the name specified in the module definition. The name of port cannot be a bit select or a part select.

The port expression must be the name used by the instantiating module and can be one of the following:

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

The port expression is optional so that the instantiating module can document the existence of the port without connecting it to anything. The parentheses are required.

The two types of module port connections can not be mixed; connections to the ports of a particular module instance must be all by order or all by name.

```
module adc4 (out, rem, in);
input in;
output [3:0] out;    output rem;
electrical [3:0] out;
electrical in, rem, rem_chain;

adc2 hi (.in(in), .out(out[3:2]), .remainder(rem_chain)) ;
adc2 lo (.in(rem_chain), .out(out[1:0]), .remainder(rem)) ;
endmodule

module adc2 (out, in, remainder);
output [1:0] out;    output remainder;
input in;
electrical [1:0] out;
electrical in, remainder, r;

adc hi1 (out[1], r, in) ; // adc is same as defined in section 7.3.3
adc lo1 (out[0], remainder, r) ;
endmodule
```

Since these connections were made by port name, the order in which the connections appear is irrelevant.

7.3.5 Port connection rules

The following rules govern the way module ports are declared and the way they are interconnected.

7.3.5.1 Compatible discipline rule

All ports connected to a node must be compatible with each other as well as to the discipline of the node. For discussion on compatible disciplines, see section 3.4.

Ports of any discipline are compatible when connected to a ground node.

7.3.5.2 Matching size rule

A scalar port can be connected to a scalar node, and a vector port can be connected to a vector node of the matching width. In other words, sizes of the ports and nodes must match.

7.3.6 Inheriting Port Natures

If a node is missing a nature, it will inherit that nature from any port that connects to it. Typically such a situation occurs when

- a node is either implicitly or explicitly declared with an empty discipline.
- a conservative port connects to a node that is declared as a signal flow discipline.
- a signal-flow port with a potential nature connects to a signal-flow node declared with a flow nature, or visa versa.

As additional ports connect to the same node, it is possible for conflicts to develop. For example, connecting either an electrical or a mechanical port to a **wire** node results in no conflicts, but connecting both to the same **wire** node does result in conflicts.

At each node there may be many different values of the absolute tolerance **abstol**. This may be because various ports connecting to the node have different, yet compatible, natures for either the potential, the flow, or both. Even if the natures are identical, the value of **abstol** may be overridden in the discipline of one or more of the ports. In such cases, all of the ablsolute tolerances must be satisfied at the node. This leads to applying the smallest tolerance value for all calculations involving such nodes.

7.3.7 Multi-disciplinary example

The example below shows how an application that spans multiple disciplines can be modeled in Verilog-A. The example models a DC-motor driven by a voltage source.

```
module motorckt();
parameter real freq=100;

electrical drive, gnd;
mechanical shaft;

motor m1 (drive, gnd, shaft);
vsource #(.freq(freq), .ampl(1.0)) v1 (drive, gnd);

endmodule

// vp:   positive terminal [V,A]    vn:     negative terminal [V,A]
// shaft:motor shaft [rad,Nm]
//
// INSTANCE parameters
// Km = motor constant [Vs/rad]    Kf = flux constant  [Nm/A]
// j  = inertia factor [Nms^2/rad] D= drag (friction)  [Nms/rad]
// Rm = motor resistance [Ohms]    Lm = motor inductance [H]
//
// A model of a DC motor driving a shaft

module motor(vp, vn, shaft);
inout vp, vn, shaft;
electrical vp, vn ;
mechanical shaft ;

parameter real Km = 4.5, Kf = 6.2;
parameter real j = .004, D = 0.1;
parameter real Rm = 5.0, Lm = .02;

analog begin
      V(vp, vn) <- Km*W(shaft) + Rm*I(vp, vn) + ddt(Lm*I(vp, vn));
      T(shaft) <- Kf*I(vp, vn) - D*W(shaft) - ddt(j*W(shaft));
end
endmodule
```

7.4 Hierarchical names

Every identifier in a Verilog-A HDL description has a unique *hierarchical path name*. The hierarchy of modules and the definition of items such as named blocks within the modules define these names. The hierarchy of names can be viewed as a tree structure, where each module instance or a named begin-end block defines a new hierarchical level, or scope, in a particular branch of the tree.

At the top of the name hierarchy are the names of modules of which no instances have been created. It is the *root* of the hierarchy. Inside any module, each module instance,

and named begin-end block define a new branch of the hierarchy. Named blocks within named blocks also create new branches.

Each node in the hierarchical name tree is treated as a separate scope with respect to identifiers. A particular identifier can be declared at most once in any scope.

Any named object can be referenced uniquely in its full form by concatenating the names of the module instance or named blocks that contain it. The period character (.) is used to separate each of the names in the hierarchy. The complete path name to any object starts at a top-level module. This path name can be used from any level in the description. The first name in a path name can also be the top of a hierarchy that starts at the level where the path is being used.

```
module samplehold (in, cntrl, out );
input in, cntrl ;
output out ;
electrical in, cntrl, out ;
electrical store, sample ;
parameter real vthresh = 0.0 ;
parameter real cap = 10e-9 ;

   amp op1 (in, sample, sample) ;
   amp op2(store, out, out) ;

   analog begin
      I(store) <+ cap * ddt(V(store)) ;
      if (V(cntrl) > vthresh)
         V(store, sample) <+ 0 ;
      else
         I(store, sample) <+ 0 ;
   end
endmodule
```

```
module amp(inp, inm, out) ;
input inp, inm ;
output out ;
electrical inp, inm, out ;
parameter real gain=1e5;

   analog begin
      V(out) <+ gain*V(inp,inm) ;
   end
endmodule
```

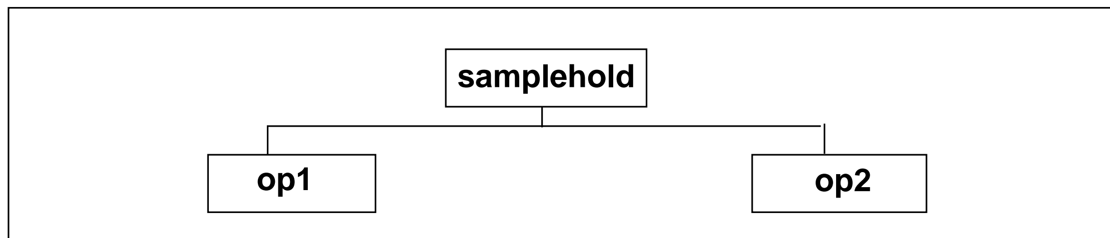


Figure 7-6: : Hierarchy in a model

```
samplehold     in, cntrl, out, sample, store, vthresh, cap
op1            op1.inp, op1.inm, op1.out, op1.gain
op2            op2.inp, op2.inm, op2.out, op2.gain
```

Figure 7-7: : Hierarchical path names in a model

From within an analog block, it is possible to use hierarchical name referencing to access signals on an external branch, but not external variables or parameters. When accessing external branches, a branch signal (its potential or flow) can be monitored (probed), or with source branches, contributions can be made to the output signal. However, contributing to an external switch branch is considered illegal.

It is illegal to indirectly assign to an external branch or contribute to an external branch that has indirect branch assignment.

7.5 Scope rules

The following two elements define a new scope in Verilog-A HDL:

```
modules
named blocks
```

An identifier can be used to declare only one item within a scope. This rule means it is illegal to declare two or more variables that have the same name, or to give an instance the same name as the name of the node connected to its output.

If an identifier is referenced directly (without a hierarchical path) within a named block, it must be declared either locally within the named block, or within a module, or named block that is higher in the same branch of the name tree that contains the named block. If it is declared locally, then the local item must be used; if not, the search continue upward until an item by that name is found or until a module boundary is encountered. The search can cross named block boundaries, but not module boundaries.

Because of the upward searching, path names that are not strictly on a downward path can be used.

Annex A
Scheduling Semantics

Analog Simulation Cycle

Simulation of a network, or system, starts with an analysis of each node to develop equations that define the complete set of values and flows in a network. Through transient analysis, the value and flow equations are solved incrementally with respect to time. At each time increment, equations for each signal are iteratively solved until they converge on a final solution.

Nodal Analysis

To describe a network, simulators combine constitutive relationships with Kirchhoff's laws in *nodal analysis* to form a system of differential-algebraic equations of the form

$$f(v, t) = \frac{dq(v, t)}{dt} + i(v, t) = 0$$
$$v(0) = v_0$$

These equations are a restatement of Kirchhoff's Flow Law.

v is a vector containing all node values

t is time

q and i are the dynamic and static portions of the flow

$f(\ )$ is a vector containing the total flow out of each node

v_0 is the vector of initial conditions

This equation was formulated by treating all nodes as being conservative (even signal flow nodes). In this way, signal-flow and conservative terminals can be connected naturally. However, this results in unnecessary KFL equations for those nodes with only signal-flow terminals attached. This situation is easily recognized and those unnecessary equations are eliminated along with the associated flow unknowns, which must be by definition zero.

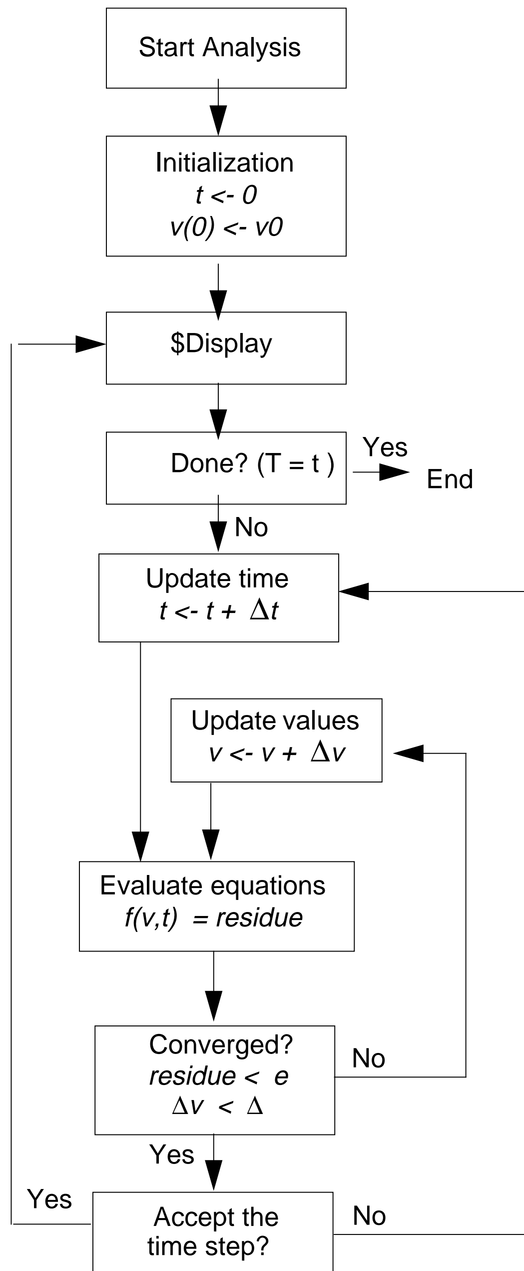
Transient Analysis

The equation describing the network is differential and nonlinear, which makes it impossible to solve directly. There are a number of different approaches to solving this

problem numerically. However, all approaches discretize time and solve the nonlinear equations iteratively.

The simulator replaces the time derivative operator (*dq/dt*) with a discrete-time finite difference approximation. The simulation time interval is discretized and solved at individual time points along the interval. The simulator controls the interval between the time points to ensure the accuracy of the finite difference approximation. At each time point, a system of nonlinear algebraic equations is solved iteratively. Most circuit simulators use the NR method to solve this system.

Simulation Flowchart (Transient Analysis)



Convergence

In Verilog-A, the behavioral description is evaluated iteratively until the NR method converges. On the first iteration, the signal values used in Verilog-A expressions are approximate and do not satisfy Kirchhoff's laws.

In fact, the initial values might not be reasonable, so you must write models that do something reasonable even when given unreasonable signal values.

For example, if you compute the log or square root of a signal value, some signal values cause the arguments to these functions to become negative, even though a real-world system never exhibits negative values.

As the iteration progresses, the signal values approach the solution. Iteration continues until two convergence criteria are satisfied. The first criterion is that the proposed solution on this iteration, $v^{(j)}(t)$, must be close to the proposed solution on the previous iteration, $v^{(j-1)}(t)$, and

$$| v_n^{(j)} - v_n^{(j-1)} | < reltol\ (max(| v_n^{(j)}| , |v_n^{(j-1)}|)) + abstol$$

where *reltol* is the relative tolerance and *abstol* is the absolute tolerance.

reltol is set as a simulator option and typically has a value of 0.001. There can be many absolute tolerances, and which one is used depends on the quantity the signal represents (volts, amps, and so on). The absolute tolerance is important when v_n is converging to zero. Without *abstol*, the iteration never converges.

The second criterion ensures that Kirchhoff's flow law is satisfied:

$$\left| \sum_n f_n (v^{(j)}) \right| < reltol\,(max\,(\left|f^i_n (v^{(j)})\right|)) + abstol$$

where $f_n^{\,i}(v^{(j)})$ is the flow exiting node *n* from branch *i*.

Both of these criteria specify the absolute tolerance to ensure that convergence is not precluded when v_n or $f_n(v)$ go to zero. While you can set the relative tolerance once in an options statement to work effectively on any node in the circuit, the absolute tolerance must be scaled appropriately for its associated signal. The absolute tolerance should be the largest signal value that is considered negligible on all the signals with which it is associated.

The simulator uses absolute tolerance to get an idea of the scale of signals. Absolute tolerances are typically 1,000 to 1,000,000 times smaller than the largest typical value for signals of a particular quantity. For example, in a typical integrated circuit, the largest potential is about 5 volts, so the default absolute tolerance for voltage is 1μV. The largest current is about 1mA, so the default absolute tolerance for current is 1pA.

Annex B
Open Issues

This appendix contains the list of all open issues known to the working group at this time:

- Array of parameters
- Array initialization
- Connect statement and matching different discipline connections
- Port branches
- Reg data type
- Initial_step, Final_step and gloabl simulator events
- Replacing behavioral models with SPICE models (SPICE compatibility)

Annex C
Syntax

This appendix contains the formal syntax definition of Verilog-A HDL. The conventions used are described in Section 1, Overview.

C.1 Source text

```
source_text ::=
      {description}
description ::=
      module_declaration
    | discipline_declaration
    | nature_declaration
module_declaration ::=
      module module_identifier [ list_of_ports ] ;
      [ module_items ]
      endmodule
list_of_ports ::=
      (  port { , port } )
port ::=
      [ port_expression ]
    | . port_identifier ( [ port_expression ] )
port_expression ::=
      port_identifier
    | port_identifier [ constant_expression ]
    | port_identifier [ msb_constant_expression :
                        lsb_constant_expression ]
module_items ::=
      { module_item }
    | analog_block
module_item ::=
      module_item_declaration
    | parameter_override
    | module_instantiation
    | analog_block
module_item_declaration ::=
      parameter_declaration
    | input_declaration
    | output_declaration
    | inout_declaration
    | integer_declaration
    | real_declaration
    | node_declaration
    | branch_declaration
```

```
parameter_override ::=
      defparam list_of_param_assignments ;
```

C.2 Natures

```
nature_declaration ::=
      nature nature_name
      [ nature_descriptions ]
      endnature
nature_name ::=
      nature_identifier
   |  nature_identifier : parent_identifier
parent_identifier ::=
      nature_identifier
   |  discipline_identifier.flow
   |  discipline_identifier.potential
nature_descriptions ::=
      nature_description
   |  nature_description nature_descriptions
nature_description ::=
   attribute = constant_expression ;
attribute ::=
      abstol
   |  access
   |  ddt_nature
   |  idt_nature
   |  units
   |  identifier
```

C.3 Disciplines

```
discipline_declaration ::=
      discipline discipline_identifier
      [ discipline_descriptions ]
      enddiscipline
discipline_descriptions ::=
      discipline_description
   |  discipline_description discipline_descriptions
discipline_description ::=
      nature_binding
   |  attr_description
nature_binding ::=
      pot_or_flow nature_identifier ;
attr_description ::=
      pot_or_flow . attribute_identifier = constant_expression ;
pot_or_flow ::=
      potential
   |  flow
```

C.4 Declarations

```
parameter_declaration ::=
      parameter [opt_type] list_of_param_assignments ;
opt_type ::=
      real
    | integer
list_of_param_assignments ::=
      declarator_init
    | list_of_param_assignments , declarator_init
declarator_init ::=
      parameter_identifier = constant_expression [ { opt_range } ]
opt_range ::=
      from range_specifier
    | exclude range_specifier
    | exclude constant_expression
range_specifier ::=
      start_paren expression1 : expression2 end_paren
start_paren ::=
      [
    | (
end_paren ::=
      ]
    | )
expression1 ::=
      constant_expression
    | -inf
expression2 ::=
      constant_expression
    | inf

input_declaration ::=
      input [range]  list_of_port_identifiers ;
output_declaration ::=
      output [range]  list_of_port_identifiers ;
inout_declaration ::=
      inout [range] list_of_port_identifiers ;
list_of_port_identifiers ::=
      port_identifier { , port_identifier }

integer_declaration ::=
      integer  list_of_identifiers ;
real_declaration ::=
      real list_of_identifiers ;
list_of_identifiers ::=
      var_name { , var_name }
var_name ::=
      variable_identifier
    | array_identifier range

node_declaration ::=
      discipline_identifier [range] list_of_nodes ;
```

```
list_of_nodes ::=
      node_identifier
   |  node_identifier , list_of_nodes

branch_declaration ::=
      branch list_of_branches ;
list_of_branches ::=
      list_of_parallel_branches
   |  list_of_parallel_branches , list_of_branches
list_of_parallel_branches ::=
      terminals list_of_branch_identifiers
terminals ::=
      ( node_identifier )
   |  ( node_identifier , node_identifier )
list_of_branch_identifiers ::=
      branch_identifier
   |  branch_identifier , list_of_branch_identifiers

block_item_declaration ::=
      parameter_declaration
   | integer_declaration
   | real_declaration
```

C.5 Module instantiation

```
module_instantiation ::=
      module_identifier [ parameter_value_assignment ] instance_list
instance_list ::=
      module_instance { , module_instance } ;
module_instance ::=
      name_of_instance  ( [ list_of_module_connections ] )
name_of_instance ::=
      module_instance_identifier [ range ]
list_of_module_connections ::=
      ordered_port_connection { , ordered_port_connection }
   |  named_port_connection { , named_port_connection }
ordered_port_connection ::=
      [ expression ]
named_port_connection ::=
      . port_identifier ( [ expression ] )
parameter_value_assignment ::=
      # ( ordered_param_override_list )
   |  # ( named_param_override_list )
ordered_param_override_list ::=
      constant_expression { , constant_expression }
named_param_override_list ::=
      named_param_override { , named_param_override }
named_param_override ::=
      . parameter_identifier ( constant_expression )
```

C.6 Behavioral statements

```
analog_block ::=
   analog statement
statement ::=
      null_statement
    | block_statement
    | branch_contribution
    | indirect_branch_assignment
    | procedural_assignment
    | conditional_statement
    | loop_statement
    | case_statement
    | generate_statement
    | event_controlled_statement
    | discontinuity_function
    | bound_step_function
    | last_crossing_function
    | system_task_enable

null_statement ::=
      ;

block_statement ::=
      begin [ : block_identifier { block_item_declaration } ]
         { statement }
      end

branch_contribution ::=
      bvalue <+ expression ;
bvalue ::=
      access_identifier ( analog_signal_list )
analog_signal_list ::=
      branch_identifier
    | node_or_port_identifier
    | node_or_port_identifier , node_or_port_identifier

indirect_branch_assignment ::=
      target : equation ;
target ::=
      bvalue
equation ::=
      nexpr == expression
nexpr ::=
      bvalue
    | ddt ( bvalue )
    | idt ( bvalue )

procedural_assignment ::=
      lexpr = expression ;
lexpr ::=
      integer_identifier
```

```
    |  real_identifier
    |  array_element
array_element ::=
       integer_identifier [ constant_expression ]
    |  real_identifier [ constant_expression ]

conditional_statement ::=
       if ( expression ) statement
       [ else statement ]

case_statement ::=
       case ( expression )
        case_item {case_item}
       endcase
case_item ::=
       expression { , expression } : statement
    |  default [ : ] statement
loop_statement ::=
       forever statement
    |  repeat ( expression ) statement
    |  while ( expression ) statement
    |  for ( procedural_assignment ; expression ;
             procedural_assignment ) statement

generate_statement ::=
       generate index_identifier ( start_expr, end_expr [, incr_expr ] )
       statement
start_expr ::=
       constant_expression
end_expr ::=
       constant_expression
incr_expr ::=
       constant_expression

event_controlled_statement ::=
       @ ( event_expression ) statement
event_expression ::=
       simple_event [ or event_expression ]
simple_event ::=
       global_event
    |  event_function
    |  identifier

global_event ::=
       initial_step [ ( analysis_list ) ]
    |  final_step [ ( analysis_list ) ]
analysis_list ::=
       analysis_name { , analysis_name }
analysis_name ::=
       " analysis_identifier "
```

```
event_function ::=
      cross_function
   |  timer_function

cross_function ::=
      cross ( expression [ , opt_args ] )
opt_args ::=
      direction [ , time_tol [ , expression_tol ] ]
direction ::=
      +1 | -1
time_tol ::=
      expression
expression_tol ::=
      expression

timer_function ::=
      timer ( start_time [ , period ] )
start_time ::=
      expression
period ::=
      expression

discontinuity_function ::=
      discontinuity ( constant_expression )

bound_step_function ::=
      bound_step ( max_step )
max_step ::=
      constant_expression

last_crossing_function ::=
      last_crossing ( expression [ , direction ] )

system_task_enable ::= system_task_name [ ( expression
                                           { , expression }  ) ] ;
system_task_name ::= $identifier
Note: The $ may not be followed by a space.
```

C.7 Expressions

```
range ::=
      [ constant_expression : constant_expression ]

constant_expression  ::=
      constant_primary
   |  string
   |  unary_operator constant_primary
   |  constant_expression binary_operator constant_expression
   |  constant_expression ? constant_expression : constant_expression
```

```
constant_primary ::=
      number
    | parameter_identifier
expression ::=
      primary
    | unary_operator primary
    | expression binary_operator expression
    | expression ? expression : expression
    | analog_function ( arg_list )
    | access_function ( arg_list )
    | built-in_function ( arg_list )
    | system_function ( arg_list )
arg_list ::=
      expression { , expression }
access_function ::=
      bvalue
unary_operator ::=
      + | - | ! | ~
binary_operator ::=
      + | - | * | / | % | == | != | && | ||
    | < | <= | > | >= | & | | | ^ | ^~ | ~^ | >> | <<
primary ::=
      number
    | identifier
    | identifier [ expression ]
    | string
    | nexpr
    | ( expression )
number ::=
      decimal_number
    | real_number
decimal_number ::=
      [ sign ] unsigned_num
real_number ::=
      [ sign ] unsigned_num . unsigned_num
    | [ sign ] unsigned_num [ . unsigned_num ] e [ sign ] unsigned_num
    | [ sign ] unsigned_num [ . unsigned_num ] E [ sign ] unsigned_num
    | [ sign ] unsigned_num [ . unsigned_num ] unit_letter
sign ::=
      +
    | -
unsigned_num ::=
      decimal_digit { _ | decimal_digit }
decimal_digit ::=
      0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
unit_letter ::=
      T | G | M | K | m | u | n | p | f | a
analog_function ::=
      ddt | idt | delay | transition | slew
    | laplace_zd | laplace_zp | laplace_np | laplace_nd
    | zi_zp | zi_zd | zi_np | zi_nd | analysis | ac_stim
    | white_noise | flicker_noise | noise_table
```

```
built_in_function ::=
      ln   | log  | exp  | sqrt  | min   | max   | abs   | pow
    | sin  | cos  | tan  | asin  | acos  | atan  | atan2
    | sinh | cosh | tanh | asinh | acosh | atanh | hypot
system_function ::=
      $limexp | $realtime | $temperature | $vt
```

C.8 General

```
comment ::=
      short_comment
    | long_comment
short_comment ::=
      // comment_text \n
long_comment ::=
      /* comment_text */
comment_text ::=
      { Any_ASCII_character  }
string ::=
      " { Any_ASCII_character_except_newline } "
identifier  ::=
      IDENTIFIER [ { . IDENTIFIER } ]
NOTE: The period in identifier may not be preceded or followed by a
space.

IDENTIFIER ::=
      simple_identifier
    | escaped_identifier
simple_identifier ::=
      [a-zA-Z]{a-zA-Z_$0-9}
escaped_identifier ::=
      \ { Any_ASCII_character_except_white_space } white_space
white_space ::=
      space
    | tab
    | newline
```

Annex D1
Keywords

This annex contains the list of all keywords used in Verilog-A HDL

```
abs
abstol
access
acos
acosh
ac_stim
always
analog
analysis
and
asin
asinh
assign
atan
atan2
atanh
begin
bound_step
branch
buf
bufif0
bufif1
case
casex
casez
cmos
cos
cosh
cross
ddt
ddt_nature
deassign
default
defparam
delay
disable
discipline
discontinuity
edge
else
end
enddiscipline
endcase
endmodule
endfunction
endnature
endprimitive
endspecify
endtable
endtask
event
exclude
exp
final_step
flicker_noise
flow
for
force
forever
fork
from
function
generate
ground
highz0
highz1
hypot
idt
idt_nature
if
ifnone
inf
initial
initial_step
inout
input
integer
join
laplace_nd
laplace_np
laplace_zd
laplace_zp
large
last_crossing
ln
log
macromodule
max
medium
min
module
nand
nature
negedge
nmos
noise_table
nor
not
notif0
notif1
or
output
parameter
pmos
posedge
potential
pow
primitive
pull0
pull1
pullup
pulldown
rcmos
real
realtime
reg
release
repeat
rnmos
rpmos
rtran
rtranif0
rtranif1
scalared
sin
sinh
slew
small
specify
specparam
sqrt
strong0
strong1
supply0
supply1
table
tan
tanh
task
temperature
time
timer
tran
tranif0
tranif1
transition
tri
tri0
tri1
triand
trior
trireg
units
vectored
vt
wait
wand
weak0
weak1
while
white_noise
wire
wor
xnor
xor
zi_nd
zi_np
zi_zd
zi_zp
```

Annex E

System Tasks and Functions

This annex describes system tasks and functions available in Verilog-A HDL.

E.1 $random function

Syntax:

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

The system function **$random** provides a mechanism for generating random numbers. The function returns a new 32-bit random number each time it is called. The random number is a signed integer; it can be positive or negative.

The `seed` parameter controls the numbers that **$random** returns. The `seed` parameter must be either a register, an integer, or a time variable. The seed value should be assigned to this variable prior to calling **$random**.

Examples:

1. Where b > 0 the expression `(`**$random %** `b)` gives a number in the following range: `[(-b+1): (b-1)]`. The following code fragment shows an example of random number generation between -59 and 59:

```
integer rand;
rand = $random % 60;
```

2. The following example shows how adding the concatenation operator to the preceding example gives `rand` a positive value from 0 to 59.

```
integer rand;
rand = {$random} % 60;
```

E.2 $dist_ functions

Syntax:

```
$dist_uniform (seed, start, end) ;
$dist_normal (seed, mean, standard_deviation) ;
$dist_exponential (seed, mean) ;
$dist_poisson (seed, mean) ;
$dist_chi_square (seed, degree_of_freedom) ;
$dist_t (seed, degree_of_freedom) ;
$dist_erlang (seed, k_stage, mean) ;
```

Figure E-1: : Syntax for the probabilistic distribution functions

All parameters to the system functions are integer values. For the exponential, poisson, chi-square, t, and erlang functions, the parameters mean, degree of freedom, and k_stage must be greater than `0`.

Each of these functions returns a pseudo-random number whose characteristics are described by the function name. That is, **$dist_uniform** returns random numbers uniformly distributed in the interval specified by its parameters.

For each system function, the seed parameter is an in-out parameter; that is, a value is passed to the function and a different value is returned. The system functions will always return the same value given the same seed. This facilitates debugging by making the operation of the system repeatable. The argument for the seed parameter should be an integer variable that is initialized by the user and only updated by the system function. This will ensure that the desired distribution is achieved.

All functions return a real value.

In the **$dist_uniform** function, the start and end parameters are integer inputs which bound the values returned. The start value should be smaller than the end value.

The mean parameter, used by **$dist_normal**, **$dist_exponential**, **$dist_poisson**, and **$dist_erlang**, is an integer input which causes the average value returned by the function to approach the value specified.

The standard deviation parameter used with the **$dist_normal** function is an integer input which helps determine the shape of the density function. Larger numbers for standard deviation will spread the returned values over a wider range. With a mean of 0 and standard deviation of 1, **$dist_normal** generates gaussian distribution.

The degree of freedom parameter used with the **$dist_chi_square** and **$dist_t** functions is an integer input which helps determine the shape of the density function. Larger numbers will spread the returned values over a wider range.

E.3 Simulation control system tasks

There are two simulation control system tasks, **$finish** and **$stop**.

E.3.1 $finish

Syntax:

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

The **$finish** system task simply makes the simulator exit and pass control back to the host operating system. If an expression is supplied to this task, then its value determines the diagnostic messages that are printed before the prompt is issued. If no argument is supplied, then a value of 1 is taken as the default.

Parameter Value	Diagnostic Message
0	prints nothing
1	prints simulation time and location
2	prints simulation time, location, and statistics about the memory and CPU time used in simulation

E.3.2 $stop

Syntax:

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

The **$stop** system task causes simulation to be suspended. This task takes an optional expression argument (0, 1, or 2) that determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of the optional argument passed to **$stop**.

E.4 File operation tasks

E.4.1 $fopen

Syntax:

```
integer  multi_channel_descriptor = $fopen ( " file_name " ) ;
```

The function **$fopen** opens the file specified as an argument and returns a 32-bit unsigned multichannel descriptor that is uniquely associated with the file. It returns 0 if the file could not be opened for writing.

The multichannel descriptor should be thought of as a set of 32 flags, where each flag represents a single output channel. The least significant bit (bit 0) of a multichannel

descriptor always refers to the standard output. The standard output is also called channel 0. The other bits refer to channels that have been opened by the **$fopen** system function.

The first call to **$fopen** opens channel 1 and returns a multichannel descriptor value of 2—that is, bit 1 of the descriptor is set. A second call to **$fopen** opens channel 2 and returns a value of 4—that is, only bit 2 of the descriptor is set. Subsequent calls to `$fopen` open channels 3, 4, 5, and so on and return values of 8, 16, 32, and so on, up to a maximum of 32 open channels. Thus, a channel number corresponds to an individual bit in a multichannel descriptor.

E.4.2 $fclose

Syntax:

```
file_close_task ::=
      $fclose ( multi_channel_descriptor ) ;
```

The **$fclose** system task closes the channels specified in the multichannel descriptor, and does not allow any further output to the closed channels. The **$fopen** task will reuse channels that have been closed.

E.5 Displaying results

The system task **`$strobe`** provides the ability to display simulation data when the simulator has converged on a solution for all nodes.

The **$strobe task** displays its arguments in the same order they appear in the argument list. Each argument can be a quoted string, an expression that returns a value, or a null argument.

The contents of string arguments are output literally except when certain escape sequences are inserted to display special characters or specify the display format for a subsequent expression.

Escape sequences are inserted into a string in three ways:

— The special character \ indicates that the character to follow is a literal or non-printable character (see Table E-1).

— The special character `%` indicates that the next character should be interpreted as a format specification that establishes the display format for a subsequent expression argument (Table E-2). For each `%` character that appears in a string, a corresponding expression argument must be supplied after the string.

— The special character string `%%` indicates the display of the percent sign character `%` (see Table E-1).

Any null argument produces a single space character in the display. (A null argument is characterized by two adjacent commas in the argument list.)

The **$strobe** task, when invoked without arguments, simply prints a newline character.

E.5.1 Escape sequences for special characters

The following escape sequences, when included in a string argument, cause special characters to be displayed:

Table E-1: Escape sequences for printing special characters

\n	is the newline character
\t	is the tab character
\\	is the \ character
\"	is the " character
\ddd	is a character specified by 1 to 3 octal digits
%%	is the % character

E.5.2 Format specifications

Table E-2 shows the escape sequences used for format specifications. Each escape sequence, when included in a string argument, specifies the display format for a subsequent expression. For each `%` character (except `%m`) that appears in a string, a corresponding expression must follow the string in the argument list. The value of the expression replaces the format specification when the string is displayed.

Table E-2: Escape sequences for format specifications

%h or %H	display in hexadecimal format
%d or %D	display in decimal format
%o or %O	display in octal format
%b or %B	display in binary format
%c or %C	display in ASCII character format
%m or %M	display hierarchical name
%s or %S	display as a string

Any expression argument that has no corresponding format specification is displayed using the default decimal format in **$strobe.**

The format specifications in Table E-3 are used with real numbers and have the full formatting capabilities available in the C language. For example, the format specification `%10.3g` specifies a minimum field width of 10 with 3 fractional digits.

Table E-3: Format specifications for real numbers

%e or %E	display 'real' in an exponential format
%f or %F	display 'real' in a decimal format
%g or %G	display 'real' in exponential or decimal format, whichever format results in the shorter printed output

E.5.3 Hierarchical name format

The `%m` format specifier does not accept an argument. Instead, it causes the display task to print the hierarchical name of the module, task, function, or named block that invokes the system task containing the format specifier. This is useful when there are many instances of the module that calls the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier will pinpoint the module instance responsible for generating the timing check message.

E.5.4 String format

The `%s` format specifier is used to print ASCII codes as characters. For each `%s` specification that appears in a string, a corresponding parameter must follow the string in the argument list. The associated argument is interpreted as a sequence of 8-bit hexadecimal ASCII codes, with each 8 bits representing a single character. If the argument is a variable, its value should be right-justified so that the right-most bit of the value is the least-significant bit of the last character in the string. No termination character or value is required at the end of a string, and leading zeros are never printed.

Annex F
Compiler Directives

All Verilog-A HDL compiler directives are preceded by the (`) character. This character is called accent grave. It is different from the character ('), which is the single quote character. The scope of compiler directives extends from the point where it is processed, across all files processed, to the point where another compiler directive supersedes it or the processing completes.

This annex describes the following compiler directives:

```
`default_nodetype
`define
`else
`endif
`ifdef
`include
`resetall
`undef
```

F.1 `default_nodetype

The directive **`default_nodetype** controls the node type created for implicit node declarations (see section 3.3.4). It can be used only outside of module definitions. It affects all modules that follow the directive, even across source file boundaries. Multiple **`default_nodetype** directives are allowed. The latest occurrence of this directive in the source controls the type of nodes that will be implicitly declared. The following is the syntax of the directive:

```
default_nodetype_compiler_directive ::=
      `default_nodetype discipline_identifier
   |  `default_nodetype wire
```

Figure F-1: Syntax for default nodetype compiler directive

When no **`default_nodetype** directive is present, implicit nodes are of type **wire**.

F.2 `define and `undef

A text macro substitution facility has been provided so that meaningful names can be used to represent commonly used pieces of text. For example, in the situation where a constant number is repetitively used throughout a description, a text macro would be useful in that only one place in the source description would need to be altered if the value of the constant needed to be changed.

F.2.1 `define

The directive **`define** creates a macro for text substitution. This directive can be used both inside and outside module definitions. After a text macro is defined, it can be used in the source description by using the (`) character, followed by the macro name. The compiler substitutes the text of the macro for the string `` `macro_name ``. All compiler directives are considered pre-defined macro names; it is illegal to re-define a compiler directive as a macro name.

A text macro can be defined with arguments. This allows the macro to be customized for each use individually.

The syntax for text macro definitions is as follows:

```
text_macro_definition ::=
      `define text_macro_name macro_text

text_macro_name ::=
      text_macro_identifier [ ( list_of_formal_arguments ) ]

list_of_formal_arguments ::=
      formal_argument_identifier { , formal_argument_identifier }
```

Figure F-2: : Syntax for text macro definition

The macro text can be any arbitrary text specified on the same line as the text macro name. If more than one line is necessary to specify the text, the newline must be preceded by a backslash (\). The first newline not preceded by a backslash will end the macro text. The newline preceded by a backslash is replaced in the expanded macro with a newline (but without the preceding backslash character).

When formal arguments are used to define a text macro, the scope of the formal arguments extend up to the end of the macro text. A formal argument can be used in the macro text in the same manner as an identifier.

If a one-line comment (that is, a comment specified with the characters //) is included in the text, then the comment does not become part of the text substituted. The macro text

can be blank, in which case the text macro is defined to be empty and no text is substituted when the macro is used.

The syntax for using a text macro is as follows:

```
text_macro_usage ::=
      `text_macro_identifier [ ( list_of_actual_arguments ) ]

list_of_actual_arguments ::=
      actual_argument { , actual_argument }

actual_argument ::=
      expression
```

Figure F-3: : Syntax for text macro usage

For an argument-less macro, the text is substituted "as is" for every occurrence of `text_macro. However, a text macro with one or more arguments must be expanded by substituting each formal argument with the expression used as the actual argument in the macro usage.

Once a text macro name has been defined, it can be used anywhere in a source description; that is, there are no scope restrictions. Text macros may be defined and used interactively.

The text specified for macro text can not be split across the following lexical tokens:

- comments
- numbers
- strings
- identifiers
- keywords
- operators

Examples:

```
`define M_PI    3.14159265358979323846

`define  size 8
electrical [1:` size] vout;

//define an adc with variable delay
`define var_adc(dly) adc #(dly)

`var_adc(2) g121 (q21, n10, n11);
`var_adc(5) g122 (q22, n10, n11);
```

The following is illegal syntax because it is split across a string:

```
`define first_half "start of string
$display(`first_half end of string");
```

Note: Text macro names can not be the same as compiler directive keywords.

Note: Text macro names can re-use names being used as ordinary identifiers. For example, `signal_name` and `` `signal_name `` are different.

Note: Redefinition of text macros is allowed; the latest definition of a particular text macro read by the compiler prevails when the macro name is encountered in the source text.

F.2.2 `undef

The directive **`undef** undefines a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a **`define** compiler directive can result in a warning. The syntax for **`undef** compiler directive is as follows:

```
undefine_compiler_directive ::=
      `undef text_macro_name
```

Figure F-4: : Syntax for undef compiler directive

An undefined text macro has no value.

F.3 `ifdef, `else, `endif

These conditional compilation compiler directives are used to optionally include lines of a Verilog-A HDL source description during compilation. The **`ifdef** compiler directive checks for the definition of a variable name. If the variable name is defined then the lines following the **`ifdef** directive are included. If the variable name is not defined and an **`else** directive exists then this source is compiled.

These directives may appear anywhere in the source description.

Situations where the **`ifdef**, **`else**, and **`endif** compiler directives may be useful include:

— selecting different representations of a module such as behavioral, structural, or mixed level
— choosing different timing or structural information
— selecting different stimulus for a given simulation run

The **`ifdef**, **`else**, and **`endif** compiler directives have the following syntax:

```
conditional_compilation_directive ::=
      `ifdef text_macro_name
         first_group_of_lines
      [ `else
         second_group_of_lines
      `endif ]
```

Figure F-5: : Syntax for conditional compilation directives

The text macro name is a Verilog-A HDL identifier. The first group of lines and the second group of lines are parts of a Verilog-A HDL source description. The **`else** compiler directive and the second group of lines are optional.

The **`ifdef**, **`else**, and **`endif** compiler directives work in the following manner:

— When an **`ifdef** is encountered, the text macro name is tested to see if it is defined as a text macro name using **`define** within the Verilog-A HDL source description.
— If the text macro name is defined, the first group_of_lines is compiled as part of the description. If there is an **`else** compiler directive, the second group of lines is ignored.
— If the text macro name has not been defined, the first group of lines is ignored. If there is an **`else** compiler directive the second group of lines is compiled.

Note: Any group of lines that the compiler ignores still must follow the Verilog-A HDL lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

Note: These compiler directives may be nested.

F.4 `include

The file inclusion (**`include**) compiler directive is used to insert the entire contents of a source file in another file during compilation. The result is as though the contents of the included source file appear in place of the **`include** compiler directive. The **`include** compiler directive can be used to include global or commonly used definitions and tasks without encapsulating repeated code within module boundaries.

Advantages of using the **`include** compiler directive include the following:

— providing an integral part of configuration management
— improving the organization of Verilog-A HDL source descriptions
— facilitating the maintenance of Verilog-A HDL source descriptions

The syntax for the **`include** compiler directive is as follows:

```
include_compiler_directive ::=
      `include "filename"
```

Figure F-6: : Syntax for include compiler directive

The compiler directive **`include** can be specified anywhere within the Verilog-A HDL description. The *filename* is the name of the file to be included in the source file. The *filename* can be a full or relative path name.

Only white space or a comment may appear on the same line as the **`include** compiler directive.

A file included in the source using **`include** compiler directive may contain other **`include** compiler directives. The number of nesting levels for included files are finite.

Examples:

Examples of legal comments for the **`include** compiler directive are as follows:

```
`include "parts/count.v"

`include "fileA"
`include "fileB" // including fileB
```

Note: Implementations may limit the maximum number of levels to which include files can be nested, but the limit shall be at least 15.

F.5 `resetall

When **`resetall** compiler directive is encountered during compilation, all compiler directives are set to the default values. This is useful for ensuring that only those directives that are desired in compiling a particular source file are active.

The recommended usage is to place **`resetall** at the beginning of each source text file, followed immediately by the directives desired in the file.

Annex G
Standard Definitions

This annex contains the standard definition package for Verilog-A HDL

```
`ifdef DISCIPLINES_H
`else
`define DISCIPLINES_H 1

//
// Natures and Disciplines
//

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

// Electrical

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

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

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

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

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

// Signal flow disciplines
discipline voltage
   potential    Voltage;
enddiscipline

discipline current
   potential    Current;
enddiscipline


// Magnetic
// Magnetomotive force in Ampere-Turns.
nature Magneto_Motive_Force
   units      = "A*turn";
   access     = MMF;
`ifdef MAGNETO_MOTIVE_FORCE_ABSTOL
   abstol     = `MAGNETO_MOTIVE_FORCE_ABSTOL;
`else
   abstol     = 1e-12;
`endif
endnature
```

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


// Thermal

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

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

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


// Kinematic

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

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

// Acceleration in meters per second squared
nature Acceleration
   units      = "m/s^2";
   access     = Acc;
   ddt_nature = Impulse;
   idt_nature = Velocity;
`ifdef ACCELERATION_ABSTOL
   abstol     = `ACCELERATION_ABSTOL;
`else
   abstol     = 1e-6;
'endif
endnature

// Impulse in meters per second cubed
nature Impulse
   units      = "m/s^3";
   access     = Imp;
   idt_nature = Acceleration;
`ifdef IMPULSE_ABSTOL
   abstol     = `IMPULSE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Force in newtons
nature Force
   units      = "n";
   access     = F;
`ifdef FORCE_ABSTOL
   abstol     = `FORCE_ABSTOL;
`else
   abstol     = 1e-6;
'endif
endnature

// Conservative disciplines
discipline kinematic
   potential    Position;
   flow         Force;
enddiscipline
```

```
discipline kinematic_v
   potential    Velocity;
   flow         Force;
enddiscipline

// Rotational

// Angle in radians
nature angle
   units      = "rads";
   access     = Theta;
   ddt_nature = Angular_Velocity;
`ifdef ANGLE_ABSTOL
   abstol     = `ANGLE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Angular Velocity in radians per second
nature Angular_Velocity
   units      = "rads/s";
   access     = Omega;
   ddt_nature = Angular_Acceleration;
   idt_nature = Angular_Velocity;
`ifdef ANGULAR_VELOCITY_ABSTOL
   abstol     = `ANGULAR_VELOCITY_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Angular acceleration in radians per second squared
nature Angular_Acceleration
   units      = "rads/s^2";
   access     = Alpha;
   ddt_nature = Angular_Velocity;
`ifdef ANGULAR_ACCELERATION_ABSTOL
   abstol     = `ANGULAR_ACCELERATION_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature

// Force in newtons
nature Angular_Force
   units      = "n/m";
   access     = Tau;
`ifdef ANGULAR_FORCE_ABSTOL
   abstol     = `ANGULAR_FORCE_ABSTOL;
`else
   abstol     = 1e-6;
`endif
endnature
```

```
// Conservative disciplines
discipline rotational
   potential    Angle;
   flow         Angular_Force;
enddiscipline

discipline rotational_omega
   potential    Angular_Velocity;
   flow         Angular_Force;
enddiscipline

`endif
```

```
// Mathematical and physical constants

`ifdef CONSTANTS_H
`else
`define CONSTANTS_H 1

// M_ is a mathmatical constant

`define     M_E           2.7182818284590452354
`define     M_LOG2E       1.4426950408889634074
`define     M_LOG10E      0.43429448190325182765
`define     M_LN2         0.69314718055994530942
`define     M_LN10        2.30258509299404568402
`define     M_PI          3.14159265358979323846
'define     M_TWO_PI      6.28318530717958647652
`define     M_PI_2        1.57079632679489661923
`define     M_PI_4        0.78539816339744830962
`define     M_1_PI        0.31830988618379067154
`define     M_2_PI        0.63661977236758134308
`define     M_2_SQRTPI    1.12837916709551257390
`define     M_SQRT2       1.41421356237309504880
`define     M_SQRT1_2     0.70710678118654752440

// P_ is a physical constant

// charge of electron in coulombs
`define     P_Q           1.6021918e-19

// speed of light in vacuum in meters/sec
`define     P_C           2.997924562e8

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

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

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

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

// zero celsius in kelvin
`define     P_CELSIUS0    273.15

'endif
```

Annex H
Glossary

Glossary of Terms

B

behavioral description

A mathematical mapping of inputs to outputs for a module, including intermediate variables and control flow.

behavioral model

A version of a module with a unique set of parameters designed to model a specific component.

block

A level within the behavioral description of a module, delimited by *begin* and *end*.

branch

A relationship between two nodes and their attached quantities within the behavioral description of a module. Each branch has two quantities, a value and a flow, with a reference direction for each.

C

component

A fundamental unit within a system that encapsulates behavior and/or structure (also known as an *element*). Modules and models might represent a single component, or a subcircuit with many components.

constitutive relationships

The essential relationships (expressions, statements) between the outputs of a module and its inputs and parameters that define the nature of the module. These relationships constitute a behavioral description.

control flow

The conditional and iterative statements controlling the behavior of a module. These statements evaluate arbitrary variables, (counters, flags, and tokens), to control the operation of different sections of a behavioral description.

child module

A module instantiated inside the behavioral description of another, “parent” module. You must have a complete definition of the child module somewhere. A child module is also known as submodule or instantiated module.

D

declaration

A definition of the properties of a variable or a node.

dynamic attributes

The characteristics of an expression whose value is derived from the evaluation of a derivative (the *dot* function). Dynamic expressions define time-dependent module behavior. Some functions cannot operate on dynamic expressions.

E

element

A fundamental unit within the system that encapsulates behavior and/or structure (also known as an *component*).

F

flow

One of the two fundamental quantities used to simulate the behavior of a system. In electrical systems, flow is current.

G

global declarations

Declarations of variables and parameters at the beginning of a behavioral description.

I

instance

Any named occurrence of an element created from a module definition. One module definition can occur in multiple instances.

instantiation

The process of creating an instance from a module definition or simulator primitive, and defining the connectivity and parameters of that instance. (Placing the instance in the circuit or system.)

K

Kirchhoff's Laws

Physical laws that define the interconnection relationships of nodes, branches, values, and flows. They specify a conservation of flow in and out of a node and a conservation of value around a loop of branches.

L

level

One block within a behavioral description, delimited by a pair of matching keywords such as begin-end, discipline-enddiscipline.

M

model

A named instance with a unique group of parameters specifying the behavior of one particular version of a module. You can use models to instantiate elements with parametric specifications different than those in the original module definition.

module

A definition of the interfaces and behavior of a component or a function.

N

NR method

Newton-Raphson method. A generalized method for solving systems of nonlinear algebraic equations by breaking them into a series of many small linear operations ideally suited for computer processing.

node

A connection point in the system, with access functions for potential and/or flow through underlying discipline.

node declaration

The statement in a module definition, identifying the names of the nodes that are associated with the module ports or are local to the module. A node declaration also identifies the discipline of the node, which in turn identifies the access functions.

P

parameter

A variable for characterizing the behavior of an instance of a module. Parameters are defined in the first section of a module, the module interface declarations, and can be specified each time a module is called in a netlist instance statement.

parameter declaration

The statement in a module definition, which defines the instance parameters of that module.

pin

An external connection point for a module (also known as a *terminal*).

potential

One of the two fundamental quantities used to simulate the behavior of a system.

primitive

A basic component that is defined entirely in terms of behavior, without reference to any other primitives. A primitive is the smallest and simplest possible portion of a simulated circuit or system.

probe

An artificial branch introduced into a circuit (or system) that does not alter its behavior, but lets the simulator to read out the potential or flow at that point.

R

reference direction

A convention for determining whether the value of a node, the flow through a branch, the value across a branch, or the flow in or out of a terminal, is positive or negative.

reference node

The global node (which equals zero value) against which all node values are measured. The reference node is ground in an electrical system.

run time binding

The conditional introduction and removal of value and flow sources during a simulation. A value source can replace a flow source and vice versa. Binding a source to a specific node or branch prevents it from going into an unknown state.

S

scope

The current nesting level of a block statement, which includes all lines of code within one set of braces in a module definition.

structural definitions

Instantiating modules inside other modules through the use of module definitions and declarations to create a hierarchical structure in the module's behavioral description.

T

terminal

An external connection point for a module (also known as a *pin* or an *analog port*).

V

Verilog-A

Analog version of Verilog HDL. A language for behavioral description of continuous-time systems that uses a syntax similar to Verilog HDL standard IEEE 1364.

Index

Symbols

!
  logical negation operator 4-1, 4-6
!=
  logical inequality operator 4-1, 4-5
$dist_ functions E-2
$dist_chi_square E-2
$dist_erlang E-2
$dist_exponential E-2
$dist_normal E-2
$dist_poisson E-2
$dist_t E-2
$dist_uniform E-2
$fclose E-4
$finish E-3
$fopen E-3
$limexp 4-22
$random E-1
$stop E-3
$strobe
  escape sequences E-5
  format specifications E-5
$transition 4-14
%
  in format specifications E-4
  modulus operator 4-1
&
  bit-wise AND operator 4-1
&&
  logical AND operator 4-1, 4-5
*
  arithmetic multiplication operator 4-1
,,
  in null expressions E-4
/
  arithmetic division operator 4-1
<
  relational less-than operator 4-1, 4-5
<+
  branch contribution operator 5-8
<<
  left shift operator 4-2, 4-7
<=
  relational less-than-or-equal operator 4-1, 4-5
==
  logical equality operator 4-1, 4-5
>
  relational greater-than operator 4-1, 4-5
>=
  relational greater-than-or-equal operator 4-1, 4-5
>>
  right shift operator 4-2, 4-7
?:
  conditional operator 4-2
@ operator 6-8
\
  for escape sequences in strings E-4
^
  bit-wise exclusive OR operator 4-1
^~
  bit-wise equivalence operator 4-2
`
  in compiler directives F-1
`default_nodetype F-1
`define F-2
`else F-4
`endif F-4
`ifdef F-4
`include F-5
`resetall F-6
`undef F-4
|
  bit-wise inclusive OR operator 4-1
||
  logical OR operator 4-1, 4-5
~
  bit-wise negation operator 4-1
~^
  bit-wise equivalence operator 4-2

A

absolute tolerance 4-12, 4-13, 4-17, A-4
abstol 3-7
AC Stimulus 4-23
Acceleration G-4
access 3-7
Access Functions 5-1
A-D converter 4-16
always procedural block 6-1
analog block 5-8
analog bus 3-12
analog operators 4-10
  restrictions 4-11
analog procedural block 6-1

analysis dependent functions 4-22
analysis function 4-22
angle G-5
Angular_Acceleration G-5
Angular_Force G-5
Angular_Velocity G-5
arithmetic operators 4-1, 4-4
- 4-4
% 4-4
* 4-4
+ 4-4
/ 4-4
arrays
of integers 3-1
of time variables 3-1
associated reference directions 1-3

B

begin-end block statement 6-4
bidirectional port 7-9
binary operators 4-2
precedence 4-2
bit-wise operators 4-6
AND 4-1
and 4-6
equivalence 4-2
exclusive nor 4-6
exclusive OR 4-1
exclusive or 4-6
inclusive OR 4-1
inclusive or 4-6
negation 4-1
unary negation 4-6
block statement
naming of 6-3
bound_step function 6-15
branch contribution operator 5-8
branch relations 5-8, 5-9
Branches 3-15
branches 1-3
built-in primitives 1-3

C

case statement 6-5
Charge G-1
comments 2-1
compatibility rules
empty discipline rule 3-13
flow compatibility rule 3-13
nature compatibility rule 3-13
nature incompatibility rule 3-13
potential compatibility rule 3-13
self rule 3-13
units value rule 3-13
Compiler directives 2-7
concatenation
of names 7-14
conditional compilation F-4
conditional operator 4-2, 4-7
conditional operator ?: 4-2
conditional statement 6-4
Connecting module ports by name 7-11
Connecting module ports by ordered list 7-10
connecting ports
by name 7-11
rules 7-11
conservative branch 3-15
conservative disciplines 3-10
conservative nodes 3-10
constant expression 4-1
constitutive relationships 1-3, A-1
contribution statements 6-2
convergence A-3
Correlated noise 4-25
cross function 6-11
Current G-1
current G-2

D

ddt operator 4-11
ddt_nature 3-7
decimal notation 2-3
default
in case statement 6-5
in if-else-if statements 6-5
Defining a function 4-25
defparam 3-3, 7-5 to 7-6
defparam statement 7-5
delay operator 4-13
diagnostic messages
from $stop and $finish E-3
discipline 3-9
disciplines
conservative 3-10
empty 3-10
signal-flow 3-10
discontinuity 6-13
discrete-time finite difference approximation A-2

E

electrical G-2
else 6-5
else statement 6-4
embedding modules 7-1, 7-3
empty disciplines 3-10
end
sequential block 6-2
endcase 6-5

enddiscipline 3-9
endfunction 4-26
endmodule 7-2
equality operators
- != 4-5
- == 4-5
- precedence 4-5

escape sequences E-4, E-5
escaped identifiers 2-5
event
- OR construct 6-9

event or 4-2
event or operator 4-7
events
- global 6-9
- monitored 6-9

exit simulator E-3
exponentiation 4-8
expression
- evaluation order 4-3

expressions 4-9
- constant 4-1

F

file inclusion F-5
filters 4-10
final_step 6-10
finite-difference approximation A-2
flicker_noise 4-24
floating-point literals 2-4
flow 1-4
flow probe 5-3
flow source 5-2
Flux G-2
for loop 6-6
Force G-4
forever loop 6-6
format specifications E-5
- ASCII character E-5
- b or B E-5
- binary E-5
- c or C E-5
- d or D E-5
- decimal E-5
- h or H E-5
- hexadecimal E-5
- hierarchical name E-5
- m or M E-5
- o or O E-5
- octal E-5
- s or S E-5
- string E-5, E-6

function 4-26
functions
- call 4-27
- definition 4-25
- distribution E-2
- probability E-2
- returning a value 4-27

G

generate statement 6-7
global events 6-9
ground 1-3

H

hierarchical path name 7-13
hierarchy
- level 7-13
- name referencing 7-13, E-5
- scope 7-13
- scope rules for naming 7-15
- top level names 7-13

hyperbolic functions 4-8

I

ideal opamp 5-9
identifiers 2-5
- escaped 2-5
- keywords 2-5

idt operator 4-12
idt_nature 3-7
if-else statement 6-4
- omitting else from nested if 6-4

If-else-if 6-5
implicit declarations F-1
implicit equations 5-5
implicit nodes 3-13
Impulse G-4
indirect branch assignement 5-10
initial procedural block 6-1
initial_step 6-10
inout port 7-9
input port 7-9
instantiation
- of modules 7-1

instantiation of modules 7-3
integer 3-1
integers
- division 4-4

interconnection relationships 1-3

J

junction diode 5-6

K

keywords 2-5
kinematic G-4
kinematic_v G-5
Kirchhoff’s Flow Law 1-4, A-1, A-4
Kirchhoff’s laws 1-3, A-1
Kirchhoff’s Value Law 1-4

L

Laplace transform filters 4-17
laplace_nd 4-19
laplace_np 4-18
laplace_zd 4-18
laplace_zp 4-17
last_crossing function 6-16
left shift operator 4-2, 4-7
lexical token
  comment 2-1
  definition of 2-1
  number 2-2
  operator 2-2
  types 2-1
  white space 2-1
limited exponential 4-22
logical operators 4-5
  ! 4-6
  && 4-5
  || 4-5
  AND 4-1
  equality 4-1
  inequality 4-1
  negation 4-1
  OR 4-1
  precedence 4-5
looping statement
  for loop 6-6
  forever loop 6-6
  repeat loop 6-6
  while loop 6-6

M

M_1_PI G-7
M_2_PI G-7
M_2_SQRTPI G-7
M_E G-7
M_LN10 G-7
M_LN2 G-7
M_LOG10E G-7
M_LOG2E G-7
M_PI G-7
M_PI_2 G-7
M_PI_4 G-7
M_SQRT1_2 G-7
M_SQRT2 G-7
M_TWO_PI G-7
magnetic G-3
Magneto_Motive_Force G-2
mathematical function 4-8
mathematical functions 4-7
minus sign(-)
  arithmetic subtraction operator 4-1
module 7-1
  definition 7-1
  instance parameter value assignment 7-6
  instantiation 7-3
  overriding parameter values 7-5 to 7-8
  parameter dependencies 7-8
  port 7-4
  terminal 7-4
  top-level 7-2
module parameter
  dependencies 7-8
  overriding values 7-5 to 7-8
modulus operator 4-1
  definition 4-4
multi-channel descriptor E-3
multi-way decisions
  case statement 6-5
  if-else-if statement 6-5

N

named blocks
  and scope 7-15
  purpose 6-3
names
  of hierarchical paths 7-13
new line character E-5
Newton-Raphson method A-3
nodal analysis A-1
node 3-5
  in hierarchical name tree 7-14
nodes 1-4, 3-12
noise 4-24
noise_table 4-24
null
  expression E-4
numbers 2-2

O

operators 4-1 to 4-7
  - 4-1
  ! 4-1, 4-6
  != 4-1, 4-5
  % 4-1
  & 4-1
  && 4-1, 4-5
  * 4-1

  - + 4-1
  - / 4-1
  - < 4-1, 4-5
  - << 4-2, 4-7
  - <= 4-1, 4-5
  - == 4-1, 4-5
  - > 4-1, 4-5
  - >= 4-1, 4-5
  - >> 4-2, 4-7
  - ?: 4-2
  - ^ 4-1
  - ^~ 4-2
  - | 4-1
  - || 4-1, 4-5
  - ~ 4-1
  - ~^ 4-2
  - analog 4-10
  - and real numbers 3-2
  - arithmetic 4-1, 4-4
  - binary 2-2, 4-2
  - bit-wise 4-6
  - bit-wise AND 4-1
  - bit-wise equivalence 4-2
  - bit-wise exclusive OR 4-1
  - bit-wise inclusive OR 4-1
  - bit-wise negation 4-1
  - conditional 2-2, 4-2, 4-7
  - definition 2-2
  - event or 4-2
  - left shift 4-2
  - left shift operator 4-7
  - logical 4-5
  - logical AND 4-1
  - logical equality 4-1
  - logical inequality 4-1
  - logical negation 4-1
  - logical OR 4-1
  - modulus 4-1
  - power 4-8
  - relational 4-1, 4-4
  - right shift 4-2
  - right shift operator 4-7
  - shift 4-7
  - time derivative 4-11
  - time integral 4-12
  - unary 2-2
- output port 7-9
- overriding module parameter values 7-5 to 7-8
  - by name 7-7
  - defparam 7-5

P

- P_C G-7
- P_CELSIUS0 G-7
- P_EPS0 G-7
- P_H G-7
- P_K G-7
- P_Q G-7
- P_U0 G-7
- parameter
  - module type 3-2
- parameter assignment by name 7-5
- parameter assignment by order 7-5
- parentheses
  - and changing operator precedence 4-3
- plus sign(+)
  - arithmetic addition operator 4-1
- port 7-8 to 7-13
  - connecting by name 7-11
  - declaration 7-9
  - definition 7-8
  - module 7-4
  - rules for connecting 7-11
- port branch 3-15
- Port Branches 5-6
- Position G-3
- potential probe 5-3
- potential source 5-2
- pow operator 4-8
- precedence
  - binary operators 4-2
  - equality operators 4-5
  - logical operators 4-5
  - relational operators 4-5
- primitives Glossary-4
- probabilistic distribution functions E-2
  - $dist_chi_square E-2
  - $dist_erlang E-2
  - $dist_exponential E-2
  - $dist_normal E-2
  - $dist_poisson E-2
  - $dist_t E-2
  - $dist_uniform E-2
  - gaussian distribution E-2
- probe 5-3
- Probes 5-3

Q

- QAM modulator 4-15
- quantities A-4

R

- real numbers 3-1 to 3-2
  - and operators 3-2
  - conversion to integers 2-4, 3-2
  - format specifications used with E-5
  - operators with real number operands 4-2
- reference direction 1-3
- reference node 1-3

relational operators 4-1, 4-4
    < 4-5
    <= 4-5
    > 4-5
    >= 4-5
    precedence 4-5
relative tolerance A-4
repeat loop 6-6
right shift operator 4-2, 4-7
rotational G-6
rotational_omega G-6

S

s
    in string display format E-6
scalar node 3-12
scientific notation 2-3
scope
    and hierarchical names 7-14
    rules 7-15
seed E-2
shift operators 4-7
    << 4-7
    >> 4-7
signal transitions 4-13
signal-flow branch 3-15
signal-flow disciplines 3-10
signal-flow nodes 3-10
sinusoidal voltage source 6-15
slew filter 4-16
slope 4-16
source branch 5-2
Sources 5-2
standard mathematical functions 4-8
standard output E-4
stochastic analysis E-2
    probabilistic distribution functions E-2
stop E-3
strings
    display format E-5, E-6
switch branch 5-2
system tasks
    for interrupting the simulator E-3
System tasks and functions 2-7

T

Temperature G-3
terminals 1-3
text macro substitutions F-2 to F-4
    and `define F-2
    definition F-2
    redefinition F-4
    with arguments F-2
thermal G-3
time derivative operator 4-11, A-2
time integral operator 4-12
timer function 6-12
Tolerances 4-11
top-level module 7-2
transient analysis A-1
transition 4-13
transition filter 4-13
transition function 4-14
tree structure
    of hierarchical names 7-13
trigonometric functions 4-8
type specification
    parameter 3-4

U

unary operators
    ! 4-6
    << 4-7
    >> 4-7
undescore character 2-2
units 3-7
User Defined Attributes 3-9
User defined functions 4-25

V

value 1-3
value range specification
    parameter 3-4
vector branch 3-16
vector node 3-12
Velocity G-4
Voltage G-2

W

Watts G-3
while loop 6-6
white space 2-1
white_noise 4-24
wire 3-10

Z

zi_nd 4-21
zi_np 4-21
zi_zd 4-20
zi_zp 4-20
Z-transform filters 4-19