

Verilog-AMS Language Reference Manual

Analog & Mixed-Signal Extensions to Verilog-HDL

Accellera

Version 2.1, January 2003

This is a stripped down version of the Verilog-AMS LRM. The material concerning VPI (Chapters 12 and 13) and Syntax (Annex A) have been removed. The full Verilog-AMS LRM is available for a fee from www.accellera.com.

Last updated on May 12, 2006. You can find the most recent version at www.designers-guide.org.

Permission to make copies, either paper or electronic, of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that the copies are complete and unmodified. To distribute otherwise, to publish, to post on servers, or to distribute to lists, requires prior written permission.



Verilog-AMS Language Reference Manual

**Analog & Mixed-Signal Extensions
to
Verilog HDL**

Version 2.1

January 20, 2003

Accellera

Copyright© 1996-2003 by Accellera International, Inc. All rights reserved.

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 Accellera.

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

Notices

The information contained in this draft manual represents the definition of the Verilog-AMS hardware description language as proposed by Accellera (Analog and Mixed-Signal TSC) as of January 20, 2003. Accellera makes no warranties whatsoever with respect to the completeness, accuracy, or applicability of the information in this draft manual to a user's requirements.

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

Accellera does not endorse any particular simulator or other CAE tool that is based on the Verilog-AMS 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 Accellera and membership enrollment can be obtained by inquiring at the address below.

Published as: Verilog-AMS Language Reference Manual
Version 2.1, January 20, 2003.

Published by: Accellera
1370 Trancas Street, #163
Napa, CA 94558
Phone: (707) 251-9977
Fax: (707) 251-9877

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.

Ramana Aisola
Andre Baguenier
Graham Bell
William Bell
Kevin Cameron
Srikanth Chandrasekaran
Ed Chang
Joe Daniels
Raphael Dorado
John Downey
Dan FitzPatrick
Vassilios Gerousis

Ian Getreu
Kim Hailey
Steve Hamm
Graham Helwig
William Hobson
Ken Kundert
Oskar Leuthold
S. Peter Liebmann
Steve Meyer
Ira Miller
Martin O'Leary
Don O'Riordan

Tom Reeder
Steffen Rochel
Jon Sanders
John Shields
James Spoto
Richard Trihy
Yatin Trivedi
Don Webber
Frank Weiler
Ian Wilson
Alex Zamfirescu
Amir Zarkesh

Table of Contents

1	Verilog-AMS introduction	1-1
1.1	Overview	1-1
1.2	Mixed-signal language features	1-2
1.3	Systems	1-3
1.3.1	Conservative systems	1-4
1.3.2	Kirchhoff's Laws	1-5
1.3.3	Natures, disciplines, and nets	1-6
1.3.4	Signal-flow systems	1-6
1.3.5	Mixed conservative/signal flow systems	1-7
1.4	Conventions used in this document	1-10
1.5	Contents	1-11
2	Lexical conventions	2-1
2.1	Lexical tokens	2-1
2.2	White space	2-1
2.3	Comments	2-2
2.4	Operators	2-2
2.5	Numbers	2-2
2.5.1	Integer constants	2-3
2.5.2	Real constants	2-3
2.5.3	Scale factors for real constants	2-4
2.6	Strings	2-5
2.6.1	String variable declaration	2-5
2.6.2	String manipulation	2-5
2.6.3	Special characters in strings	2-6
2.7	Identifiers, keywords, and system names	2-6
2.7.1	Escaped identifiers	2-7
2.7.2	Keywords	2-7
2.7.3	System tasks and functions	2-7
2.7.4	Compiler directives	2-8
3	Data types	3-1
3.1	Integer and real data types	3-1
3.2	Parameters	3-2
3.2.1	Type specification	3-4
3.2.2	Value range specification	3-4
3.2.3	Parameter arrays	3-5
3.3	Genvars	3-5
3.4	Net_discipline	3-6
3.4.1	Natures	3-7

3.4.2	Disciplines	3-10
3.4.3	Net discipline declaration	3-15
3.4.4	Ground declaration	3-17
3.4.5	Implicit nets	3-17
3.5	Real net declarations	3-18
3.6	Default discipline	3-19
3.6.1	Disciplines of primitives	3-19
3.7	Discipline precedence	3-20
3.8	Net compatibility	3-20
3.8.1	Discipline and Nature Compatibility	3-21
3.9	Branches	3-23
3.10	Namespace	3-24
3.10.1	Nature and discipline	3-24
3.10.2	Access functions	3-25
3.10.3	Net	3-25
3.10.4	Branch	3-25

4 Expressions4-1

4.1	Operators	4-1
4.1.1	Operators with real operands	4-2
4.1.2	Binary operator precedence	4-3
4.1.3	Expression evaluation order	4-4
4.1.4	Arithmetic operators	4-4
4.1.5	Relational operators	4-5
4.1.6	Case equality operators	4-6
4.1.7	Logical equality operators	4-6
4.1.8	Logical operators	4-6
4.1.9	Bit-wise operators	4-7
4.1.10	Shift operators	4-8
4.1.11	Conditional operator	4-8
4.1.12	Event or	4-8
4.1.13	Concatenations	4-8
4.2	Built-in mathematical functions	4-9
4.2.1	Standard mathematical functions	4-9
4.2.2	Transcendental functions	4-10
4.2.3	Error handling	4-11
4.3	Signal access functions	4-11
4.4	Analog operators	4-12
4.4.1	Restrictions on analog operators	4-12
4.4.2	Vector or array arguments to analog operators	4-13
4.4.3	Analog operators and equations	4-13
4.4.4	Time derivative operator	4-13
4.4.5	Time integral operator	4-14
4.4.6	Circular integrator operator	4-15

4.4.7	Absolute delay operator	4-16
4.4.8	Transition filter	4-17
4.4.9	Slew filter	4-21
4.4.10	last_crossing function	4-22
4.4.11	Laplace transform filters	4-23
4.4.12	Z-transform filters	4-25
4.4.13	Limited exponential	4-28
4.4.14	Constant versus dynamic arguments	4-29
4.5	Analysis dependent functions	4-29
4.5.1	Analysis	4-30
4.5.2	AC stimulus	4-31
4.5.3	Noise	4-31
4.6	User-defined functions	4-33
4.6.1	Defining an analog function	4-33
4.6.2	Returning a value from an analog function	4-35
4.6.3	Calling an analog function	4-35
5	Signals	5-1
5.1	Analog signals	5-1
5.1.1	Access functions	5-1
5.1.2	Probes and sources	5-2
5.1.3	Examples	5-3
5.1.4	Port branches	5-5
5.1.5	Switch branches	5-6
5.1.6	Unassigned sources	5-7
5.2	Signal access for vector branches	5-7
5.2.1	Accessing net and branch signals	5-9
5.2.2	Accessing attributes	5-10
5.3	Contribution statements	5-10
5.3.1	Branch contribution statements	5-10
5.3.2	Indirect branch assignments	5-13
6	Analog behavior	6-1
6.1	Analog procedural block	6-1
6.2	Block statements	6-2
6.2.1	Sequential blocks	6-2
6.2.2	Block names	6-3
6.3	Procedural assignments	6-3
6.4	Conditional statement	6-4
6.4.1	Examples	6-4
6.4.2	Analog conditional statements	6-5
6.5	Case statement	6-5
6.5.1	Analog case statements	6-6
6.5.2	Constant expression in case statement	6-7

6.6	Looping statements	6-7
6.6.1	Repeat and while statements	6-7
6.6.2	For statements	6-8
6.7	Events	6-9
6.7.1	Event detection	6-9
6.7.2	Event OR operator	6-10
6.7.3	Event triggered statements	6-11
6.7.4	Global events	6-11
6.7.5	Monitored events	6-13
7	Hierarchical structures.	7-1
7.1	Modules	7-1
7.1.1	Top-level modules	7-3
7.1.2	Module instantiation	7-3
7.2	Overriding module parameter values	7-6
7.2.1	Defparam statement	7-6
7.2.2	Module instance parameter value assignment by order	7-8
7.2.3	Module instance parameter value assignment by name	7-8
7.2.4	Parameter dependence	7-9
7.3	Ports	7-9
7.3.1	Port association	7-9
7.3.2	Port declarations	7-10
7.3.3	Real valued ports	7-11
7.3.4	Connecting module ports by ordered list	7-12
7.3.5	Connecting module ports by name	7-13
7.3.6	Port connection rules	7-14
7.3.7	Inheriting port natures	7-14
7.4	Hierarchical names	7-15
7.5	Scope rules	7-16
8	Mixed signal.	8-1
8.1	Introduction	8-1
8.2	Fundamentals	8-2
8.2.1	Domains	8-2
8.2.2	Contexts	8-2
8.2.3	Nets, nodes, ports, and signals	8-2
8.2.4	Mixed-signal and net disciplines	8-4
8.3	Behavioral interaction	8-4
8.3.1	Accessing discrete nets and variables from a continuous context	8-5
8.3.2	Accessing X and Z bits of a discrete net in a continuous context	8-6
8.3.3	Accessing continuous nets and variables from a discrete context	8-8
8.3.4	Detecting discrete events in a continuous context	8-9
8.3.5	Detecting continuous events in a discrete context	8-10
8.3.6	Concurrency	8-11

8.3.7	Function calls	8-12
8.4	Discipline resolution	8-12
8.4.1	Compatible discipline resolution	8-12
8.4.2	Connection of discrete-time disciplines	8-13
8.4.3	Connection of continuous-time disciplines	8-13
8.4.4	Resolution of mixed signals	8-14
8.5	Connect modules	8-17
8.6	Connect module descriptions	8-18
8.7	Connect specification statements	8-19
8.7.1	Connect module auto-insertion statement	8-20
8.7.2	Discipline resolution connect statement	8-22
8.7.3	Parameter passing attribute	8-23
8.7.4	connect_mode	8-23
8.8	Automatic insertion of connect modules	8-23
8.8.1	Connect module selection	8-25
8.8.2	Signal segmentation	8-27
8.8.3	connect_mode parameter	8-29
8.8.4	Rules for driver-receiver segregation and connect module selection and insertion	8-33
8.8.5	Instance names for auto-inserted instances	8-34
8.9	Driver-receiver segregation	8-36
8.10	Driver access and net resolution	8-38
8.10.1	\$driver_count	8-39
8.10.2	\$driver_state	8-39
8.10.3	\$driver_strength	8-39
8.10.4	driver_update	8-40
8.10.5	Receiver net resolution	8-40
8.10.6	Connect module example using driver access functions	8-41
8.11	Supplementary driver access functions	8-43
8.11.1	\$driver_delay	8-43
8.11.2	\$driver_next_state	8-43
8.11.3	\$driver_next_strength	8-44
8.11.4	\$driver_type	8-44
9	Scheduling semantics	9-1
9.1	Introduction	9-1
9.2	Analog simulation cycle	9-1
9.2.1	Nodal analysis	9-1
9.2.2	Transient analysis	9-2
9.2.3	Convergence	9-3
9.3	Mixed-signal simulation cycle	9-4
9.3.1	Circuit initialization	9-5
9.3.2	Synchronization of analog and digital in transient analysis	9-5
9.3.3	The synchronization loop	9-10

9.3.4	Synchronization and communication algorithm	9-13
9.3.5	Assumptions about the analog and digital algorithms	9-14
9.4	Scheduling semantics for the digital engine	9-15
9.4.1	The stratified event queue	9-15
9.4.2	The Verilog-AMS digital engine reference model	9-16
9.4.3	Scheduling implication of assignments	9-17
10	System tasks and functions	10-1
10.1	Environment parameter functions	10-1
10.2	\$random function	10-2
10.3	\$dist_ functions	10-2
10.4	Simulation control system tasks	10-4
10.4.1	\$finish	10-4
10.4.2	\$stop	10-4
10.5	File operation tasks	10-5
10.5.1	\$fopen	10-5
10.5.2	\$fclose	10-5
10.6	Display tasks	10-5
10.6.1	Escape sequences for special characters	10-6
10.6.2	Format specifications	10-7
10.6.3	Hierarchical name format	10-8
10.6.4	String format	10-8
10.7	Announcing discontinuity	10-8
10.8	Time related functions	10-10
11	Compiler directives.....	11-1
11.1	`default_discipline	11-1
11.2	`default_transition	11-2
11.3	`define and `undef	11-3
11.3.1	`define	11-3
11.3.2	`undef	11-5
11.4	`ifdef, `else, `endif	11-5
11.5	`include	11-6
11.6	`resetall	11-7
11.7	Predefined macros	11-8
12	Using VPI routines	12-1
12.1	The VPI interface	12-1
12.1.1	VPI callbacks	12-1
12.1.2	VPI access to Verilog-AMS HDL objects and simulation objects	12-2
12.1.3	Error handling	12-2
12.2	VPI object classifications	12-2
12.2.1	Accessing object relationships and properties	12-3

12.2.2	Delays and values	12-4
12.3	List of VPI routines by functional category	12-5
12.4	Key to object model diagrams	12-7
12.4.1	Diagram key for objects and classes	12-8
12.4.2	Diagram key for accessing properties	12-9
12.4.3	Diagram key for traversing relationships	12-10
12.5	Object data model diagrams	12-11
12.5.1	Module	12-12
12.5.2	Nature, discipline	12-13
12.5.3	Scope, task, function, IO declaration	12-14
12.5.4	Ports	12-15
12.5.5	Nodes	12-16
12.5.6	Branches	12-17
12.5.7	Quantities	12-18
12.5.8	Nets	12-19
12.5.9	Regs	12-20
12.5.10	Variables, named event	12-21
12.5.11	Memory	12-22
12.5.12	Parameter, specparam	12-23
12.5.13	Primitive, prim term	12-24
12.5.14	UDP	12-25
12.5.15	Module path, timing check, intermodule path	12-26
12.5.16	Task and function call	12-27
12.5.17	Continuous assignment	12-28
12.5.18	Simple expressions	12-29
12.5.19	Expressions	12-30
12.5.20	Contribs	12-31
12.5.21	Process, block, statement, event statement	12-32
12.5.22	Assignment, delay control, event control, repeat control	12-33
12.5.23	While, repeat, wait, for, forever	12-34
12.5.24	If, if-else, case	12-35
12.5.25	Assign statement, deassign, force, release, disable	12-36
12.5.26	Callback, time queue	12-37
13	VPI routine definitions	13-1
13.1	vpi_chk_error()	13-3
13.2	vpi_compare_objects()	13-4
13.3	vpi_free_object()	13-5
13.4	vpi_get()	13-6
13.5	vpi_get_cb_info()	13-7
13.6	vpi_get_analog_delta()	13-8
13.7	vpi_get_analog_freq()	13-9
13.8	vpi_get_analog_time()	13-10
13.9	vpi_get_analog_value()	13-11

13.10	vpi_get_delays()	13-13
13.11	vpi_get_str()	13-16
13.12	vpi_get_analog_systf_info()	13-17
13.13	vpi_get_systf_info()	13-18
13.14	vpi_get_time()	13-19
13.15	vpi_get_value()	13-20
13.16	vpi_get_vlog_info()	13-26
13.17	vpi_get_real()	13-27
13.18	vpi_handle()	13-28
13.19	vpi_handle_by_index()	13-29
13.20	vpi_handle_by_name()	13-30
13.21	vpi_handle_multi()	13-31
	13.21.1 Derivatives for analog system task/functions	13-31
	13.21.2 Examples	13-31
13.22	vpi_iterate()	13-35
13.23	vpi_mcd_close()	13-37
13.24	vpi_mcd_name()	13-38
13.25	vpi_mcd_open()	13-39
13.26	vpi_mcd_printf()	13-40
13.27	vpi_printf()	13-41
13.28	vpi_put_delays()	13-42
13.29	vpi_put_value()	13-45
13.30	vpi_register_cb()	13-47
	13.30.1 Simulation-event-related callbacks	13-48
	13.30.2 Simulation-time-related callbacks	13-50
	13.30.3 Simulator analog and related callbacks	13-51
	13.30.4 Simulator action and feature related callbacks	13-51
13.31	vpi_register_analog_systf()	13-53
	13.31.1 System task and function callbacks	13-54
	13.31.2 Declaring derivatives for analog system task/functions	13-54
	13.31.3 Examples	13-55
13.32	vpi_register_systf()	13-59
	13.32.1 System task and function callbacks	13-59
	13.32.2 Initializing VPI system task/function callbacks	13-61
13.33	vpi_remove_cb()	13-62
13.34	vpi_scan()	13-63
13.35	vpi_sim_control()	13-64

A	Syntax.....	A-1
A.1	Source text	A-1
A.2	Natures	A-2
A.3	Disciplines	A-3
A.4	Declarations	A-3
A.5	Module instantiation	A-5

A.6	Mixed signal	A-6
A.7	Behavioral statements	A-6
A.8	Analog expressions	A-9
A.9	Expressions	A-9
A.10	General	A-12
B	Keywords	B-1
B.1	All keywords	B-1
B.2	Discipline/nature	B-3
B.3	Connect rules	B-3
C	Analog language subset	C-1
C.1	Verilog-AMS introduction	C-1
C.1.1	Verilog-A overview	C-1
C.1.2	Verilog-A language features	C-1
C.2	Lexical conventions	C-2
C.3	Data types	C-2
C.4	Expressions	C-3
C.5	Signals	C-3
C.6	Analog behavior	C-3
C.7	Hierarchical structures	C-3
C.8	Mixed signal	C-4
C.9	Scheduling semantics	C-4
C.10	System tasks and functions	C-4
C.11	Compiler directives	C-4
C.12	Using VPI routines	C-4
C.13	VPI routine definitions	C-4
C.14	Syntax	C-5
C.15	Keywords	C-5
C.16	Standard definitions	C-5
C.17	SPICE compatibility	C-5
C.18	Changes from previous Verilog-A LRM versions	C-6
C.19	Obsolete functionality	C-9
C.19.1	Forever	C-9
C.19.2	NULL	C-9
C.19.3	Generate	C-9
D	Standard definitions.	D-1
D.1	The disciplines.vams file	D-2
D.2	The constants.vams file	D-8
D.3	The driver_access.vams file	D-9
E	SPICE compatibility	E-1

E.1	Introduction	E-1
E.1.1	Scope of compatibility	E-1
E.1.2	Degree of incompatibility	E-1
E.2	Accessing Spice objects from Verilog-AMS HDL	E-2
E.2.1	Case sensitivity	E-2
E.2.2	Examples	E-3
E.3	Preferred primitive, parameter, and port names	E-4
E.3.1	Independent sources	E-5
E.3.2	Unsupported components	E-6
E.3.3	Discipline of primitives	E-6
E.4	Other issues	E-7
E.4.1	Multiplicity factor on subcircuits	E-7
E.4.2	Binning and libraries	E-7
F	Discipline resolution methods	F-1
F.1	Discipline resolution	F-1
F.2	Resolution of mixed signals	F-1
F.2.1	Default discipline resolution algorithm	F-1
F.2.2	Alternate expanded analog discipline resolution algorithm	F-2
G	Open issues	G-1
H	Glossary	H-1

Section 1

Verilog-AMS introduction

1.1 Overview

This Verilog-AMS Hardware Description Language (HDL) language reference manual defines a behavioral language for analog and mixed-signal systems. Verilog-AMS HDL is derived from *IEEE 1364-1995 Verilog HDL*. This document is intended to cover the definition and semantics of Verilog-AMS HDL as proposed by Accellera.

Figure 1-1 shows the components and architecture of Verilog-AMS HDL, which consists of the complete *IEEE 1364-1995 Verilog HDL* specification (noted as Verilog-D in the figure), an analog equivalent for describing analog systems (noted as Verilog-A), and extensions to both for specifying the full Verilog-AMS HDL (noted as MS Extensions).

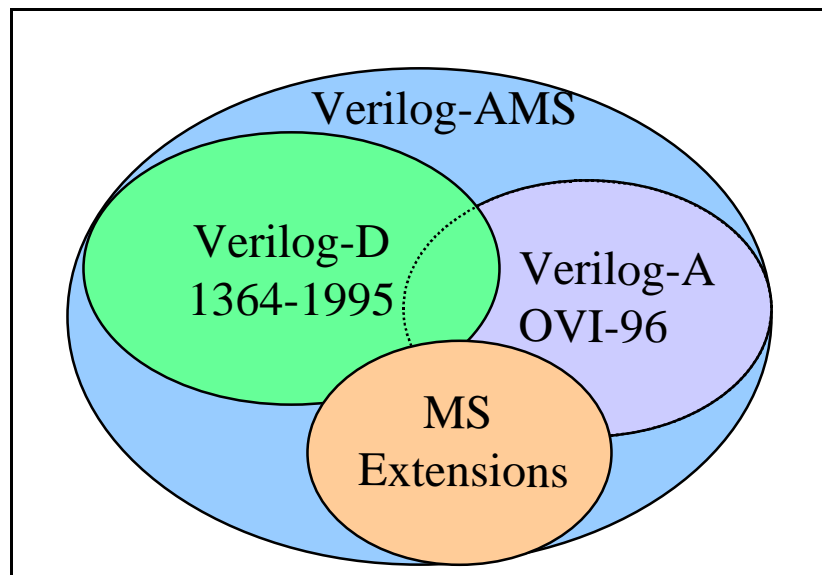


Figure 1-1 Verilog-AMS architecture

Verilog-AMS HDL lets designers of analog and mixed-signal systems and integrated circuits create and use modules which 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 ports 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.

For continuous systems, Verilog-AMS HDL is defined to be applicable to both electrical and non-electrical systems description. It supports *conservative* and *signal-flow* descriptions by using the concepts of *nets*, *nodes*, *branches*, and *ports* as terminology for these descriptions. 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 (e.g., voltage and current) associated with the analog behaviors.

Verilog-AMS HDL can also be used to describe discrete (digital) systems (per *IEEE 1364-1995 Verilog HDL*) and mixed-signal systems using both discrete and continuous descriptions as defined in this LRM.

1.2 Mixed-signal language features

Verilog-AMS HDL extends the features of the digital modeling language (*IEEE 1364-1995 Verilog HDL*) to provide a single unified language with both analog and digital semantics with backward compatibility. Below is a list of salient features of the resulting language:

- signals of both analog and digital types can be declared in the same module
- `initial`, `always`, and `analog` procedural blocks can appear in the same module
- both analog and digital signal values can be accessed (read operations) from any context (analog or digital) in the same module
- digital signal values can be set (write operations) from any context outside of an `analog` procedural block
- analog potentials and flows can only receive contributions (write operations) from inside an `analog` procedural block
- the semantics of the `initial` and `always` blocks remain the same as in *IEEE 1364-2001 Verilog HDL*; the semantics for the `analog` block are described in this manual
- the `discipline` declaration is extended to digital signals
- a new construct, `connect` statement, is added to facilitate auto-insertion of user-defined connection modules between the analog and digital domains
- when hierarchical connections are of mixed type (i.e., analog signal connected to digital port or digital signal connected to analog port), user-defined connection modules are automatically inserted to perform signal value conversion

1.3 Systems

A *system* is considered to be a collection of interconnected *components* which are acted upon by a stimulus and produce a response. The components themselves can also be systems, in which case a *hierarchical system* is defined. If a component does not have any subcomponents, it is considered to be a *primitive component*. Each primitive component connects to zero or more nets. Each net connects to a signal which can traverse multiple levels of the hierarchy. The behavior of each component is defined in terms of values at each net.

A *signal* is a hierarchical collection of nets which, because of port connections, are contiguous. If all the nets which make up a signal are in the discrete domain, the signal is a *digital signal*. If, on the other hand, all the nets which make up a signal are in the continuous domain, the signal is an *analog signal*. A signal which consists of nets from both domains is called a *mixed signal*.

Similarly, a port whose connections are both analog is an *analog port*, a port whose connections are both digital is a *digital port*, and a port whose connections are both analog and digital is a *mixed port*. The components connect to nodes through ports and nets to build a hierarchy, as shown in Figure 1-2.

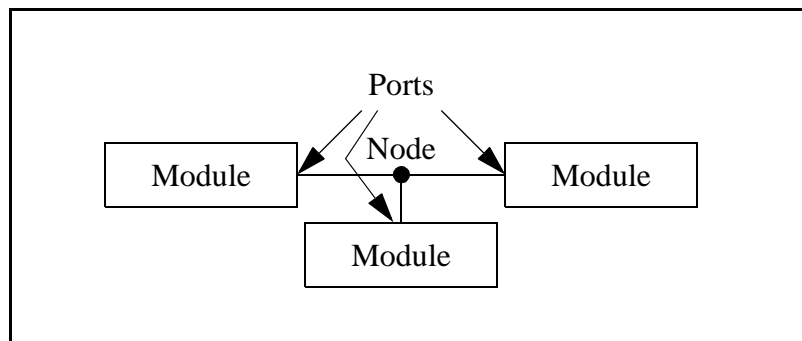


Figure 1-2 Components connect to nodes through ports

If a signal is analog or mixed, it is associated with a node (see Section 3.4), while a purely digital signal is not associated with a node. Regardless of the number of analog nets in an analog or mixed signal or how the analog nets in a mixed signal are interspersed with digital nets, the analog portion of an analog or mixed signal is represented by only a single node. This guarantees a mixed or analog signal has only one value which represents its potential with respect to the global reference voltage (*ground*).

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 usually given structurally. That is, the description of a system contains instances of components and how they are interconnected. Descriptions of components are given using behavior and or structure. A behavior is a mathematical description which relates the signals at the ports of the components.

1.3.1 Conservative systems

An important characteristic of conservative systems is there are two values associated with every node, the *potential* (also known as the *across value* or *voltage* in electrical systems) and the *flow* (the *through value* or *current* in electrical systems). The potential of the node is shared with all continuous ports and nets connected to the node so all continuous ports and nets see the same potential. The flow is shared so flow from all continuous ports and nets at a node shall sum to zero (0). 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 port or net, it can either affect, or be affected by, either the potential at the node, and/or the flow onto the node through the port or net.

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.

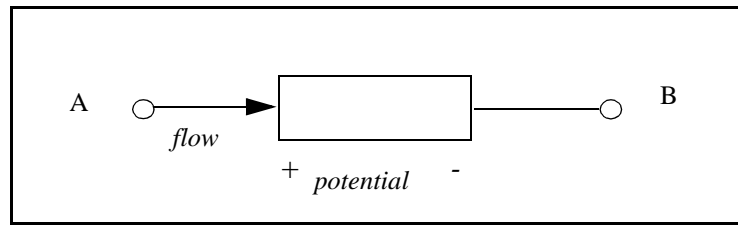
A behavioral description of a conservative component is constructed as a collection of interconnected branches. The constitutive equations of the component are formulated as to relate the branch potentials and flows. In the probe/source approach (see Section 5.1.2), the branch potential or flow is specified as a function of branch potentials and flows. If the branch potential and flow are left unspecified, not on the left-hand side of a contribution statement, 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 (0). Otherwise the branch flow is assumed to be zero (0) 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 which are useless and confusing.)

1.3.1.1 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 (0). Any net of continuous discipline can be declared to be **ground**. In this case, the node associated with the net shall be the global reference node in the circuit. This is compatible with all analog disciplines and can be used to bind a port of an instantiated module to the reference node.

1.3.1.2 Reference directions

The reference directions for a generic branch are shown in Figure 1-3.

**Figure 1-3 Reference directions**

The *reference direction* for a potential is indicated by the plus and minus symbols near each port. Given the chosen reference direction, the branch potential is positive whenever the potential of the port marked with a plus sign (A) is larger than the potential of the port 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-AMS HDL uses associated reference directions. A positive flow enters a branch through the port marked with the plus sign and exits the branch through the port marked with the minus sign.

1.3.2 Kirchhoff's Laws

In formulating continuous system equations, Verilog-AMS HDL uses two sets of relationships. The first are the constitutive relationships which 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-AMS 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.

A Verilog-AMS 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, as shown in Figure 1-4.

- Kirchhoff's Flow Law (KFL)
The algebraic sum of all flows out of a node at any instant is zero (0).
- Kirchhoff's Potential Law (KPL)
The algebraic sum of all the branch potentials around a loop at any instant is zero (0).

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

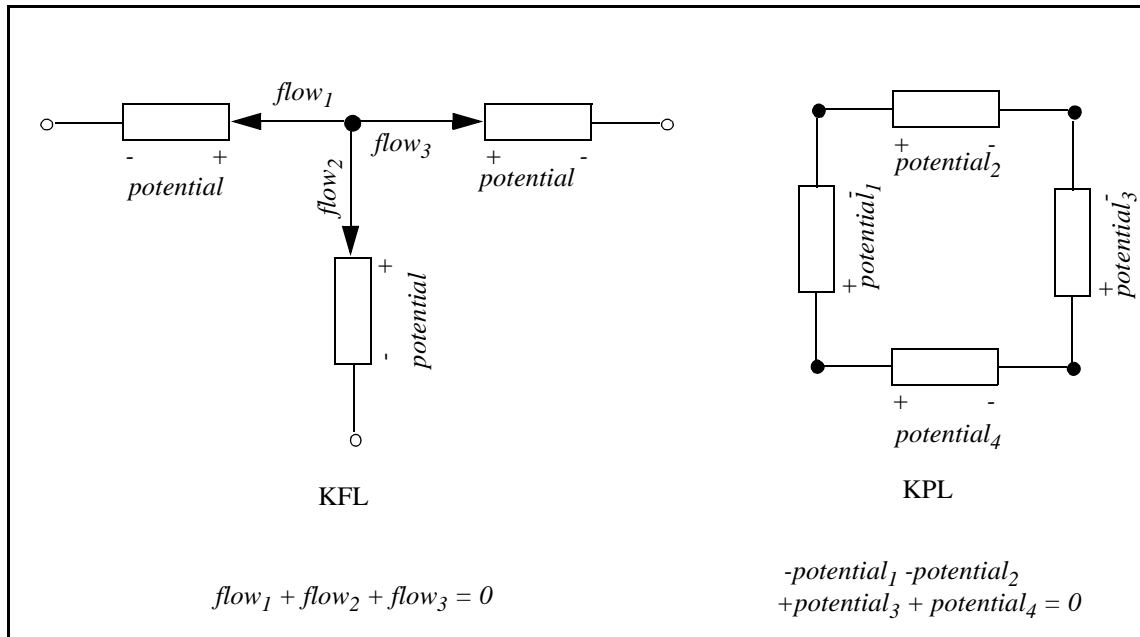


Figure 1-4 Kirchhoff's Flow Law (KFL) and Potential Law (KPL)

1.3.3 Natures, disciplines, and nets

Verilog-AMS HDL allows definition of nets 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, which 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 nets. The compatibility rules help enforce the legal operations between nets of different disciplines.

1.3.4 Signal-flow systems

A discipline may specify two nature bindings, `potential` and `flow`, or it may specify only a single binding, `potential`. Disciplines with two natures are known as *conservative disciplines* because nodes which are bound to them exhibit Kirchhoff's Flow Law, and thus, conserve charge (in the electrical case). A discipline with only a potential nature is known as a *signal flow discipline*.

As a result of port connections of analog nets, a single node may be bound to a number of nets of different disciplines. If a node is bound only to disciplines which have potential

nature only, current contributions to that node are not legal. Flow for such a node is not defined.

Signal flow models may be written so potentials of module outputs are purely functions of potentials at the inputs without taking flow into account.

The following example is a level shifting voltage follower:

```

module shiftPlus5(in, out);
input in;
output out;
voltage in, out; //voltage is a signal flow
                  //discipline compatible with
                  //electrical, but having a
                  //potential nature only

analog begin
    V(out) <+ 5.0 + V(in);
end
endmodule

```

If a number of such modules were cascaded in series, it would not be necessary to conserve charge (i.e., sum the flows) at any intervening node.

If, on the other hand, the output of this device were bound to a node of a conservative discipline (e.g., `electrical`), then the output of the device would appear to be a controlled voltage source to ground at that node. In that case, the flow (i.e., current) through the source would contribute to charge conservation at the node. If the input of this device were bound to a node of a conservative discipline then the input would act as a voltage probe to ground. Thus, when a net of signal flow discipline with potential nature only is bound to a conservative node, contributions made to that net behave as voltage sources to ground.

Nets of signal flow disciplines in modules may only be bound to `input` or `output` ports of the module, not to `inout` ports. Potential contributions may not be made to `inputs`.

1.3.5 Mixed conservative/signal flow 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 minimum changes. 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 port and net declarations only require types for those values which are actually used in the description. Thus, signal-flow ports only require the type of potential to be

specified, whereas conservative ports require types for both values (the potential and flow).

Examples:

For example, consider a differential voltage amplifier, a differential current amplifier, and a resistor. The amplifiers are written using signal-flow ports and the resistor uses conservative ports.

```

module voltage_amplifier (out, in) ;
input in ;
output out ;
voltage out ,      // Discipline voltage defined elsewhere
        in ;      // with 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 ports 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 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 ports.

```

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 ports. Both are defined in the conservative discipline `electrical`.

In summary, only those signals types declared on the ports 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 ports, without forcing types to be declared for unused signals on signal-flow nets and ports. 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 which is hidden from the user. The simulator begins by treating all ports as being conservative, which allows the connection of signal-flow and conservative ports. This results in additional unnecessary equations for those nodes which only have signal-flow ports. 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 ports, such as the ability to access flow and the ability to access floating potentials, are available with signal-flow ports.
- Natures only have to be given for potentials and flows if they are accessed in a behavioral description.
- If nets and ports are used only in a structural description (only in instance statements), then no natures need be specified.

1.4 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 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-AMS 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 the braces appear in bold face, in which case it stands for itself. The item can 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 which 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.

1.5 Contents

This document contains the following sections and annexes:

1. Verilog-AMS introduction
This section gives the overview of analog modeling, defines basic concepts, and describes Kirchhoff's Potential and Flow Laws.
2. Lexical conventions
This section defines the lexical tokens used in Verilog-AMS HDL.
3. Data types
This section describes the data types: integer, real, parameter, nature, discipline, and net, used in Verilog-AMS HDL.
4. Expressions
This section describes expressions, mathematical functions, and time domain functions used in Verilog-AMS 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-AMS HDL for behavioral modeling.
7. Hierarchical structures
This section describes how to build hierarchical descriptions using Verilog-AMS HDL.
8. Mixed signal
This section describes the mixed-signal aspects of the Verilog-AMS HDL language.
9. Scheduling semantics
This section describes the basic simulation cycle as applicable to Verilog-AMS HDL.
10. System tasks and functions
This section describes the system tasks and functions in Verilog-AMS HDL.

11. Compiler directives

This section describes the compiler directives in Verilog-AMS HDL.

12. Using VPI routines

This section describes how the VPI routines are used.

13. VPI routine definitions

This section defines each of the VPI routines in alphabetical order.

A. Syntax

This annex describes formal syntax for all Verilog-AMS HDL constructs in Backus-Naur Form (BNF).

B. Keywords

This annex lists all the words which are recognized in Verilog-AMS HDL as keywords.

C. Analog language subset

This annex describes the analog subset of Verilog-AMS HDL.

D. Standard definitions

This annex provides the definitions of several natures, disciplines, and constants which are useful for writing models in Verilog-AMS HDL.

E. SPICE compatibility

This annex describes the SPICE compatibility with Verilog-AMS HDL.

F. Discipline resolution methods

This annex provides the semantics for two methods of resolving the discipline of undeclared interconnect.

G. Open issues

This annex lists unresolved issues.

H. Glossary

This annex describes various terms used in this document.

Section 2

Lexical conventions

This section describes the lexical tokens used in Verilog-AMS HDL source text and their conventions. This section is based on Section 2, Lexical conventions, of *IEEE 1364-1995 Verilog HDL*. The changes specific to Verilog-AMS HDL can be found in Section 2.5.3 and Section 2.7.2.

2.1 Lexical tokens

A Verilog-AMS HDL source text file 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—spaces and newlines are not syntactically significant other than being token separators, except escaped identifiers (see Section 2.7.1).

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 token type contains the characters for spaces, tabs, newlines, and formfeeds. These characters are ignored except when they serve to separate other lexical tokens.

2.3 Comments

Verilog-AMS HDL has two forms to introduce comments, as shown in Syntax 2-1. A *one-line comment* starts with the two characters `//` and ends with a newline. *Block comments* start 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.

```
comment ::=
    short_comment
  | long_comment

short_comment ::=
    // { any_ASCII_characters_except_end_of_line } \n

long_comment ::=
    /* { any_ASCII_characters } */

comment_text ::=
    { Any_ASCII_character }
```

Syntax 2-1—Syntax for comments

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 which separate three operands.

2.5 Numbers

Constant numbers can be specified as integer constants or real constants. The syntax for constants is shown in Syntax 2-2.

```

number ::=
    decimal_number
    | digital_octal_number
    | digital_binary_number
    | digital_hex_number
    | real_number

decimal_number ::=
    [ sign ] unsigned_num

sign ::=
    + | -

unsigned_num ::=
    decimal_digit { _ | decimal_digit }

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

digital_number ::=
    number

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 ] scale_factor

scale_factor ::=
    T | G | M | K | k | m | u | n | p | f | a

```

Syntax 2-2—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 (64-bits).

Real numbers can be specified in either decimal notation (e.g., 14.72) or in scientific notation (e.g., 39e8, which indicates 39 multiplied by 10 to the 8th power). Real numbers

expressed with a decimal point shall have at least one digit on each side of the decimal point. The underscore character is legal anywhere in a real constant except as the first character of the constant or the first character after the decimal point. The underscore character is ignored.

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 Scale factors for real constants

Floating-point numbers can be specified using the following letter symbols for the scale factors indicated. Scale factors and scientific notation are not allowed to be used together in describing a real number.

Largest	————→	Smallest
T = 10 ¹²		m = 10 ⁻³
G = 10 ⁹		u = 10 ⁻⁶
M = 10 ⁶		n = 10 ⁻⁹
K = 10 ³ ; k = 10 ³		p = 10 ⁻¹²
		f = 10 ⁻¹⁵
		a = 10 ⁻¹⁸

No space is permitted between the number and the symbol. Scale factors are not allowed to be used in defining digital delays (e.g., #5u).

This form of floating-point number specification is provided in Verilog-AMS HDL in addition to the two methods for writing floating-point numbers described in Section 2.5.2.

Example:

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

2.6 Strings

A *string* is a sequence of characters enclosed by double quotes (“”) and contained on a single line. Strings used as operands in expressions and assignments shall be treated as unsigned integer constants represented by a sequence of 8-bit ASCII values, with one 8-bit ASCII value representing one character.

2.6.1 String variable declaration

String variables are variables of type `reg` (see *IEEE 1364-1995 Verilog HDL*) with the width equal to the number of characters in the string multiplied by eight (8).

Example:

To store the twelve-character string “Hello world!” requires a `reg` $8 * 12$ or 96 bits wide

```
reg [8*12:1] stringvar;
initial begin
    stringvar = "Hello world!";
end
```

2.6.2 String manipulation

Strings can be manipulated using the Verilog HDL operators. The value being manipulated by the operator is the sequence of 8-bit ASCII values.

Example:

```
module string_test;
reg [8*14:1] stringvar;
initial begin
    stringvar = "Hello world";
    $display("%s is stored as %h", stringvar,stringvar);
    stringvar = {stringvar,"!!!"};
    $display("%s is stored as %h", stringvar,stringvar);
end
endmodule
```


The output is:

```
Hello world is stored as 00000048656c6c6f20776f726c64
Hello world!!! is stored as 48656c6c6f20776f726c64212121
```

Note: When a variable is larger than required to hold a value being assigned, the contents on the left are padded with zeros (0) after the assignment. This is consistent with the padding which occurs during assignment of nonstring values. If a string is larger than the destination string variable, the string is truncated to the left and the leftmost characters are lost.

2.6.3 Special characters in strings

Certain characters can only be used in strings when preceded by an introductory character called an *escape character*. Table 2-1 lists these characters in the right-hand column, with the escape sequence that represents the character in the left-hand column.

Table 2-1—Specifying special characters in string

Escape string	Character produced by escape string
\n	New line character
\t	Tab character
\\	\ character
\"	" character
\ddd	A character specified in 1–3 octal digits (0 ≤ d ≤ 7)

2.7 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
```

2.7.1 Escaped identifiers

Escaped identifiers start with the backslash character (\) and end with white space (space, tab, newline, or formfeed). 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)
```

2.7.2 Keywords

Keywords are predefined non-escaped identifiers which are used to define the language constructs. Preceding a keyword with an escape character (\) causes it to be interpreted as an escaped identifier.

All keywords are defined in lowercase only. Annex A lists all defined Verilog-AMS HDL keywords (including those from *IEEE 1364-1995 Verilog HDL*).

2.7.3 System tasks and functions

The \$ character introduces a language construct which 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 given in Syntax 2-3.

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

list_of_arguments ::=
    argument { , [ argument ] }

argument ::=
    expression
```

Syntax 2-3—Syntax for system tasks and functions

Section 10 lists the standard system tasks and functions for Verilog-AMS HDL.

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;
```

2.7.4 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.

Section 11 lists the Verilog-AMS HDL compiler directives and their syntax.

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-AMS HDL supports `integer`, `real`, and `parameter` data types as found in *IEEE 1364-2001 Verilog HDL*. It also modifies the `parameter` data types and introduces *array of real* as an extension of the `real` data type. Plus, it extends the `net` data types to support a new type called `wreal` to model real value nets.

Verilog-AMS HDL introduces a new data type, called *net_discipline*, for representing analog nets and declaring *disciplines* of all `nets` and `regs`. The *disciplines* define the domain and the natures of `potential` and `flow` and their associated attributes for *continuous* domains. A new data type called *genvar* is also introduced for use with behavioral loops.

3.1 Integer and real data types

The syntax for declaring `integer` and `real` is shown in Syntax 3-1.

```
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 array_range
array_range ::=
    [ upper_limit_constant_expression : lower_limit_constant_expression ]
```

Syntax 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 $2^{31}-1$. Arrays of integers can be declared using a range which defines the upper and lower indices of the array. Both indices shall be constant expressions and shall evaluate to a positive integer, a negative integer, or zero (0).

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, as described by *IEEE STD-754-1985*, an IEEE standard for double precision floating point numbers.

Arrays of real can be declared using a range which defines the upper and lower indices of the array. Both indices shall be constant expressions and shall evaluate to a positive integer, a negative integer, or zero (0).

Integers are initialized at the start of a simulation depending on how they are used. Integer variables whose values are assigned in an analog context default to an initial value of zero (0). Integer variables whose values are assigned in a digital context default to an initial value of x. Real variables are initialized to zero (0) at the start of a 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
```

3.2 Parameters

The syntax for parameter declarations is shown in Syntax 3-2.

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

For parameters defined as arrays, the initializer shall be a `constant_param_arrayinit` expression which is a list of constant expressions containing only constant numbers and previously defined parameters within { and } delimiters.

Parameters represent constants, hence it is illegal to modify their value at runtime. However, parameters can be modified at compilation time to have values which 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. It is not legal to use hierarchical name referencing (from within the analog block) to access external analog variables or parameters.

```

parameter_declaration ::=
    parameter [opt_type] list_of_param_assignments ;

opt_type ::=
    real
    | integer

list_of_param_assignments ::=
    declarator_init { , declarator_init }

declarator_init ::=
    parameter_identifier = constant_expression { opt_value_range }
    | parameter_array_identifier range = constant_param_arrayinit { opt_value_range }

opt_value_range ::=
    from value_range_specifier
    | exclude value_range_specifier
    | exclude value_constant_expression

value_range_specifier ::=
    start_paren expression1 : expression2 end_paren

start_paren ::=
    [ | (

end_paren ::=
    ] | )

expression1 ::=
    constant_expression | -inf

expression2 ::=
    constant_expression | inf

constant_param_arrayinit ::=
    { param_arrayinit_element_list }

param_arrayinit_element_list ::=
    param_arrayinit_element { , param_arrayinit_element }

param_arrayinit_element ::=
    constant_expression
    | { replicator_constant_expression { constant_expression } }

```

Syntax 3-2—Syntax for parameter declaration

By nature, analog behavioral specifications are characterized more extensively in terms of parameters than their digital counterparts. There are three fundamental extensions to the parameter declarations defined in *IEEE 1364-1995 Verilog HDL*:

- An optional type for the parameter can be specified in Verilog-AMS HDL. In *IEEE 1364-1995 Verilog HDL*, 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 1364-1995 Verilog HDL*, this check had to be done in user's model or was left as an implementation specific detail.
- Parameter arrays of basic integer and real data types can be specified.

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 shall be specified.

Examples:

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 as in *IEEE 1364-1995 Verilog HDL*.

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 (see Section 4.1.1.1).

Example:

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

Here, `size` is coerced to 10.0.

3.2.2 Value range specification

A parameter declaration can contain optional specifications of the permissible range of the values of a parameter. More than one range can 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 shall be numerically smaller than the second expression in the range.

Examples:

```
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 default value for `neg_rail` is -15 and it is only allowed to acquire values within the range of $-50 \leq \text{neg_rail} < 0$. Similarly, the default value for parameter `pos_rail` is 15 and it is only allowed to acquire values within the range of

$0 < \text{pos_rail} < 50$. And, the default value for `gain` is 1 and it is allowed to acquire values within the range of $1 \leq \text{gain} \leq 1000$.

The keyword **inf** can be used to indicate infinity. If preceded by a negative sign, it indicates negative infinity.

Example:

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

A single value can be excluded from the possible valid values for a parameter.

Example:

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

Here, the value of a parameter is checked against the specified range. Range checking applies to the value of the parameter for the instance and not against the default values specified in the device. It shall be an error only if the value of the parameter is out of range during simulation.

3.2.3 Parameter arrays

Verilog-AMS HDL includes behavioral extensions which utilize arrays. It requires these arrays be initialized in their definitions and allow overriding their values as with other parameter types. The declaration of arrays of parameters is in a similar manner to those of parameters and register arrays of reals and integers in *IEEE 1364-1995 Verilog HDL*.

Parameter arrays have the following restrictions. Failure to follow these restrictions shall result in an error.

- A type of a parameter array shall be given in the declaration.
- An array assigned to an instance of a module shall be of the exact size of the array bounds of that instance.
- If the array size is changed via a parameter assignment, the parameter array shall be assigned an array of the new size from the same module as the parameter assignment that changed the parameter array size.

Example:

```
parameter real poles[0:3] = { 1.0, 3.198, 4.554, 2.00 };
```

3.3 Genvars

Genvars are integer-valued variables which compose static expressions for instantiating structure behaviorally such as accessing analog signals within behavioral looping constructs. The syntax for declaring genvar variables is shown in Syntax 3-3.


```

genvar_declaration ::=
    genvar list_of_genvar_identifiers ;

list_of_genvar_identifiers ::=
    genvar_identifier { , genvar_identifier }

```

Syntax 3-3—Syntax for genvar declaration

The static nature of genvar variables is derived from the limitations upon the contexts in which their values can be assigned.

Examples:

```

genvar i;
analog begin
    ...
    for (i = 0; i < 8; i = i + 1) begin
        V(out[i]) <+ transition(value[i], td, tr);
    end
    ...
end

```

The genvar variable *i* can only be assigned within the for-loop control. Assignments to the genvar variable *i* can consist only of expressions of static values, e.g., parameters, literals, and other genvar variables.

3.4 Net_discipline

In addition to the data types supported by *IEEE 1364-1995 Verilog HDL*, an additional data type, *net_discipline*, is introduced in Verilog-AMS HDL for continuous time and mixed-signal simulation. *net_discipline* is used to declare analog nets, as well as declaring the domains of digital nets and regs.

A signal can be digital, analog, or mixed, and is a hierarchical collection of nets which are contiguous (because of port connections). For analog and mixed signals, a single node is associated with all continuous nets segments of the signal. The fundamental characteristic of analog and mixed signals is the values of the associated node are determined by the simultaneous solution of equations defined by the instances connected to the node using Kirchhoff's conservation laws. In general, a node represents a point of physical connections between nets of continuous-time description and it obeys conservation-law semantics.

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

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

3.4.1 Natures

A *nature* is a collection of attributes. In Verilog-AMS HDL, there are several pre-defined attributes. In addition, user-defined attributes can 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 shown in Syntax 3-4.

```

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_description ::=
    attribute = constant_expression ;

attribute ::=
    abstol
    | access
    | ddt_nature
    | idt_nature
    | units
    | attribute_identifier

```

Syntax 3-4—Syntax for nature declaration

A nature shall be defined between the keywords **nature** and **endnature**. Each nature definition shall have a unique identifier as the name of the nature and shall include all the required attributes specified in 3.4.1.2.

Examples:

```

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

nature voltage
    units = "V" ;
    access = V;
    abstol = 1u ;
endnature

```

3.4.1.1 Derived natures

A nature can be derived from an already declared nature. This allows the new nature to have the same attributes as the attributes of the existing 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, it is called a *base nature*.

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

A derived nature can declare additional attributes or override attribute values of the parent nature, with certain restrictions (as outlined in Section 3.4.1.2) for the predefined attributes.

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

Examples:

```

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

// An alias
nature Ttl_net_curr : Ttl_curr
endnature

nature New_curr : Ttl_curr // derived, but different
    abstol = 1m ;// modified for this nature
    maxval = 12.3 ;// new attribute for this nature
endnature

```

3.4.1.2 Attributes

Attributes define the value of certain quantities which characterize the nature. There are five predefined attributes — **abstol**, **access**, **idt_nature**, **ddt_nature**, and **units**. In addition, user-defined attributes can be defined in a nature (see Section 3.4.1.3). Attribute

declaration assigns a constant expression to the attribute name, as shown in the example in Section 3.4.1.1.

abstol

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

This attribute is required for all base natures. It is legal for a derived nature to change **abstol**, but if left unspecified it shall inherit the **abstol** from its parent nature. The constant expression assigned to it shall evaluate to a real value.

access

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

This attribute is required for all 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. If specified, the constant expression assigned to it shall be an identifier (by name, not as a string).

idt_nature

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

idt_nature can be used to reduce the need to specified tolerances on the **idt()** operator. If this operator is applied directly on nets, the tolerance can be taken from the node, which eliminates the need to give a tolerance with the operator.

If specified, the constant expression assigned to **idt_nature** shall be the name (not a string) of a nature which is defined elsewhere. It is possible for a nature to be self-referencing with respect to its **idt_nature** attribute. In other words, the value of **idt_nature** can be the nature that the attribute itself is associated with.

The **idt_nature** attribute is optional; the default value is the nature itself. While it is possible to override the parent's value of **idt_nature** using a derived nature, the nature thus specified shall be related (share the same base nature) to the nature the parent uses for its **idt_nature**.

ddt_nature

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

ddt_nature can be used to reduce the need to specified tolerances on the **ddt()** operator. If this operator is applied directly on nets, the tolerance can be taken from the node, eliminating the need to give a tolerance with the operator.

If specified, the constant expression assigned to `ddt_nature` shall be the name (not a string) of a nature which is defined elsewhere. It is possible for a nature to be self-referencing with respect to its `ddt_nature` attribute. In other words, the value of `ddt_nature` can be the nature that the attribute itself is associated with.

The **ddt_nature** attribute is optional; the default value is the nature itself. While it is possible to override the parent's value of `ddt_nature` using a derived nature, the nature thus specified shall be related (share the same base nature) to the nature the parent uses for its `ddt_nature`.

units

The **units** attribute provides a binding between the value of the access function and the units for that value. The **units** field is provided so simulators can annotate the continuous signals with their units and is also used in the *net compatibility rule check*.

This attribute is required for all base natures. It is illegal for a derived nature to define or change the `units`; the derived nature always inherits its parent nature `units`. If specified, the constant expression assigned to it shall be a string.

3.4.1.3 User-defined attributes

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

A user-defined attribute can be declared in the same manner as any predefined attribute. The name of the attribute shall be unique in the nature being defined and the value being assigned to the attribute shall be constant.

3.4.2 Disciplines

A *discipline* description consists of specifying a domain type and binding any *natures* to potential or flow.

The syntax for declaring a discipline is shown in Syntax 3-5.

```

discipline_declaration ::=
    discipline discipline_identifier
    [ discipline_descriptions ]
    enddiscipline

discipline_descriptions ::=
    discipline_description { discipline_description }

discipline_description ::=
    nature_binding
    | domain_binding
    | attr_override
    | attribute_identifier = constant_expression;

nature_binding ::=
    pot_or_flow nature_identifier ;

pot_or_flow ::=
    potential
    | flow

domain_binding ::=
    domain continuous ;
    | domain discrete ;

attr_override ::=
    pot_or_flow . attribute_identifier = constant_expression ;

```

Syntax 3-5—Syntax for discipline declaration

A *discipline* shall be defined between the keywords **discipline** and **enddiscipline**. Each discipline shall 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. Analog behavioral nets (nodes) must have a discipline defined for them but interconnect and digital nets do not. It is possible to set the discipline of interconnect and digital nets through discipline declaration with hierarchical references to these nets. It shall be an error to hierarchically override the discipline of a net that was explicitly declared unless it is a compatible discipline.

3.4.2.1 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 which obeys Kirchhoff's Potential Law (KPL). This access function is called the *potential access function*.

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

Disciplines with two natures are called *conservative disciplines* and the nets associated with conservative disciplines are called *conservative nets*. Conservative disciplines shall not have the same *nature* specified for both the `potential` and the `flow`. Disciplines with a single potential nature are called *signal-flow disciplines* and the nets with signal-flow disciplines are called *signal-flow nets*. Only a `potential` nature is allowed to be specified for a signal-flow discipline, as shown in the following examples.

Examples:

Conservative discipline

```
discipline electrical
  potential Voltage ;
  flow Current ;
enddiscipline
```

Signal-flow disciplines

```
discipline voltage
  potential Voltage ;
enddiscipline

discipline current
  potential Current ;
enddiscipline
```

Multi-disciplinary example

Disciplines in Verilog-AMS HDL allow designs of multi-disciplinaries to be easily defined and simulated. Disciplines can be used to allow unique tolerances based on the size of the signals and outputs displayed in the actual units of the discipline. This example shows how an application spanning multiple disciplines can be modeled in Verilog-AMS HDL. It models a DC-motor driven by a voltage source.

```
module motorckt;
  parameter real freq=100;
  ground gnd;

  electrical drive;
  rotational 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 ;
    rotational 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*Theta(shaft) + Rm*I(vp, vn) +
            ddt(Lm*I(vp, vn));
        Tau(shaft) <+ Kf*I(vp, vn) - D*Theta(shaft) -
            ddt(j*Theta(shaft));
    end
endmodule

```

3.4.2.2 Domain binding

Analog signal values are represented in continuous time, whereas digital signal values are represented in discrete time. The **domain** attribute of the discipline stores this property of the signal. It takes two possible values, **discrete** or **continuous**. Signals with continuous-time domains are real valued. Signals with discrete-time domains can either be binary (0, 1, x, or z), integer or real values.

Examples:

```

discipline electrical
    domain continuous;
    potential Voltage;
    flow Current;
enddiscipline

discipline logic
    domain discrete ;
enddiscipline

```

The **domain** attribute is optional. The default value for domain is **continuous** for non-empty disciplines, e.g., those which specify nature bindings.

3.4.2.3 Empty disciplines

It is possible to define a discipline with no nature bindings. These are known as *empty disciplines* and they can be used in structural descriptions to let the components connected to a net determine which natures are to be used for the net.

Such disciplines may have a domain binding or they may be domain-less, thus allowing the domain to be determined by the connectivity of the net (see Section 8.4).

Example:

```
discipline interconnect
    domain continuous;
enddiscipline
```

3.4.2.4 Discipline of wires and undeclared nets

It is possible for a module to have nets where there are no discipline declarations. If such a net appears bound only to ports in module instantiations, it may have no declaration at all or may be declared to have a wire type such as `wire`, `tri`, `wand`, `wor`, etc. If it is referenced in behavioral code, then it must have a wire type.

In these cases, the net shall be treated as having an empty discipline. If the net is referenced in behavioral code, then it shall be treated as having empty discipline with a domain binding of `discrete`, otherwise it shall be treated as having empty discipline with no domain binding. If a net has a wiretype but is not connected to behavioral code (`interconnect`) and it resolved to domain `discrete` then its wiretype shall be used in any nettype resolution steps per IEEE 1364-2001.

This allows *netlists* (modules that describe connectivity only, with no behavior) that use wire as an `interconnect` to be valid in both *IEEE 1364-1995 Verilog HDL* and Verilog-AMS HDL. The domain shall be determined by the connectivity of the net (see Section 8.4).

3.4.2.5 Overriding nature attributes from discipline

A discipline can override the value of the bound nature for the pre-defined attributes (except as restricted by Section 3.4.1.2), as shown for the flow `t1l_curr` in the example below. To do so from a discipline declaration, the bound nature and attribute needs to be defined, as shown for the `abstol` value within the discipline `t1l` in the example below. The general form is shown as the *attr_override* terminal in Syntax 3-5: the keyword **flow** or **potential**, then the hierarchical separator `.` and the attribute name, and, finally, set all of this equal to (=) the new value (e.g., `flow.abstol = 10u`).

Examples:

```
nature t1l_curr
    units = "A" ;
    access = I ;
    abstol = 1u ;
endnature

nature t1l_volt
    units = "V" ;
    access = V;
    abstol = 100u ;
endnature
```

```

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

```

3.4.2.6 Deriving natures from disciplines

A nature can be derived from the *nature* bound to the `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 `potential` or the `flow` of the discipline.

If the nature binding to the `potential` or the `flow` of a discipline changes, the new nature shall 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 shall also include the discipline name followed by the hierarchical separator (`.`) and the keyword **flow** or **potential**, as shown for `Ttl_net_curr` in the example below.

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

Examples:

```

nature Ttl_net_curr : Ttl.flow // from the example in Section 3.4.2.5
endnature // abstol = 10u as modified in ttl

nature Ttl_net_volt : Ttl.potential // from the example in
// Section 3.4.2.5
  abstol = 1m ;// modified for this nature
  maxval = 12.3 ;// new attribute for this nature
endnature

```

3.4.2.7 User-defined attributes

Like natures, a discipline can specify user-defined attributes. Discipline user-defined attributes are useful for the same reasons as nature user-defined attributes (see Section 3.4.1.3).

3.4.3 Net discipline declaration

Each *net_discipline* declaration associates nets and regs with an already declared discipline. Syntax 3-6 shows how to declare disciplines of nets and regs.

```

net_discipline_declaration ::=
    discipline_identifier [ range ] list_of_nets ;
range ::=
    [ constant_expression : constant_expression ]

net_type ::=
    net_identifier [range] [= constant_expression | constant_array_expression]

list_of_nets ::=
    net_type {, net_type}

```

Syntax 3-6—Syntax for net discipline declaration

If a range is specified for a net, the net is called a *vector net*; otherwise it is called a *scalar net*. A vector net is also called an *bus*.

Examples:

```

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

```

Nets represent the abstraction of information about signals. As with ports, nets represent component interconnections. Nets declared in the module interface define the ports to the module (see Section 7.3.4).

A net used for modeling a conservative system shall have a discipline with both access functions (`potential` and `flow`) defined. When modeling a signal-flow system, the discipline of a net can have only `potential` access functions. When modeling a discrete system, the discipline of a net can only have a `domain of discrete` defined.

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

3.4.3.1 Net Discipline Initial (Nodeset) Values

Nets with continuous disciplines are allowed to have initializers on their net discipline declarations; however, nets of non-continuous disciplines are not.

Examples:

```

electrical a = 5.0;
electrical [0:4] bus = {2.3,4.5,,6.0};
mechanical top.foo.w = 250.0;

```

The initializer will be used as a nodeset value for the potential of the net by the analog solver. In the case of analog buses, a constant array expression is used as an initializer.

A null value in the constant array indicates that no nodeset value is being specified for this element of the bus.

If different nets of a node have conflicting initializers, then initializers on hierarchical net declarations win. If there are multiple hierarchical declarations, then the declaration on the highest level wins. If there are multiple hierarchical declarations on the highest level, then it is a race condition for which the initializer wins. If the multiple conflicting initializers are not hierarchical, then it is also a race condition for which the initializer wins.

3.4.4 Ground declaration

Each ground declaration is associated with an already declared net of continuous discipline. The node associated with the net will be the global reference node in the circuit. If used in behavioral code, the net shall be used in only the differential source and probe forms, e.g., `V(gnd)` is not allowed. The net must be assigned a continuous discipline to be declared ground.

Syntax 3-7 shows the syntax used for declaring the global reference node (*ground*).

```
ground_declaration ::=
    ground [ range ] list_of_nets;
```

Syntax 3-7—Syntax for declaring ground

Examples:

```
module loadedsrc(in, out);
    input in;
    output out;
    electrical in, out;
    electrical gnd;
    ground gnd;
    parameter real srcval = 5.0;

    resistor #(.r(10K)) r1(out,gnd);
    analog begin
        V(out) <+ V(in,gnd)*2;
    end
endmodule
```

3.4.5 Implicit nets

Nets can be used in a structural descriptions without being declared. In this case, the net is implicitly declared to be a scalar net with the empty discipline and undefined domain.

Examples:

```

module top(i1, i2, o1, o2, o3);
  input i1, i2;
  output o1, o2, o3;
  electrical i1, i2, o1, o2, o3;

  // ab1, ab2, cb1, cb2 are implicit nets, not declared
  blk_a a1( i1, ab1 );
  blk_a a2( i2, ab2 );
  blk_b b1( ab1, cb1 );
  blk_b b2( ab2, cb2 );
  blk_c c1( o1, o2, o3, cb1, cb2 );
endmodule

```

3.5 Real net declarations

The **wreal**, or real net data type, represents a real-valued physical connection between structural entities. A **wreal** net shall not store its value. A **wreal** net can be used for real-valued nets which are driven by a single driver, such as a continuous assignment. If no driver is connected to a **wreal** net, its value shall be zero (0.0). Unlike other digital nets which have an initial value of 'x', **wreal** nets shall have an initial value of zero.

wreal nets can only be connected to compatible interconnect and other **wreals** or real expressions. They cannot be connected to any other wires, although connection to explicitly declared 64-bit wires can be done via system tasks \$realtobits and \$bitstoreal. Compatible interconnect are nets of type wire, tri, and **wreal** where the IEEE 1364-2001 net resolution is extended for **wreal**. When the two nets connected by a port are of net type **wreal** and wire/tri, the resulting single net will be assigned as **wreal**. Connection to other nettypes will result in an error.

Syntax 3-8 shows the syntax for declaring digital nets.

```

digital_net_declaration ::=
  digital_net_declaration
  | wreal [range] list_of_identifiers ;

```

Syntax 3-8—Syntax for declaring digital nets

Examples:

```

module foo(in, out);
  input in;
  output out;
  wreal in;
  electrical out;
  analog begin
    V(out) <+ in;
  end
endmodule

module top();
  real stim;
  electrical load;
  wreal wrstim;
  assign wrstim = stim;
  foo fl(wrstim, load);
  always begin
    #1 stim = stim + 0.1;
  end
endmodule

```

3.6 Default discipline

Verilog-AMS HDL supports the ``default_discipline` compiler directive. This directive specifies a default discrete discipline to be applied to any discrete net which does not have an explicit discipline declaration. A description and its syntax are shown in Section 11.1.

3.6.1 Disciplines of primitives

With internal simulator primitives the discipline of the vpiLoConn to be used in discipline resolution during a mixed-signal simulation must be known. For digital primitives the domain is discrete and thus the discipline is set via the `default_discipline` directive as it is for digital modules. If the discipline of digital connections (vpiLoConn) to a mixed net are unknown then the `default_discipline` must be specified (via the directive or other vendor specific method). If not specified, an error will result during discipline resolution.

For analog primitives, the discipline will be defined by the attribute `port_discipline` on that instance. If no attribute is found then it will acquire the discipline of other compatible continuous disciplines connected to that net segment. If no disciplines are connected to that net, then the default discipline is set to electrical. This is further described in section E.3.3.

3.7 Discipline precedence

While a net itself can be declared only in the module to which it belongs, the discipline of the net can be specified in a number of ways.

- The discipline name can appear in the declaration of the net.
- The discipline name can be used in a declaration which makes an out of context reference to the net from another module.
- The discipline name can be used in a ``default_discipline` compiler directive.

Discipline conflicts can arise if more than one of these methods is applied to the same net. Discipline conflicts shall be resolved using the following order of precedence:

1. A declaration from a module other than the module to which the net belongs using an out of module reference, e.g.,

```
module example1;
    electrical example2.net;
endmodule
```

2. The local declaration of the net in the module to which it belongs, e.g.,

```
module example2;
    electrical net;
endmodule
```

3. ``default_discipline` with qualifier, e.g.,

```
`default_discipline logic trireg;
```

4. ``default_discipline` without qualifier, e.g.,

```
`default_discipline logic;
```

It is not legal to have two different disciplines at the same level of precedence for the same net.

3.8 Net compatibility

Certain operations can be done on nets only if the two (or more) nets are compatible. For example, if an access function has two nets as arguments, they must be compatible. The following rules shall apply to determine the compatibility of two (or more) nets:

Discrete Domain Rule: Digital nets with the same signal value type (i.e., bit, real, integer) are compatible with each other if their disciplines are compatible (i.e., the discipline has a discrete domain or is empty).

Signal Domain Rule: It shall be an error to connect two ports or nets of different domains unless there is a connect statement (see Section 8.4) defined between the disciplines of the nets or ports.

Signal Connection Rule: It shall be an error to connect two ports or nets of the same domain with incompatible disciplines.

3.8.1 Discipline and Nature Compatibility

The following rules shall apply to determine discipline compatibility:

Self Rule (Discipline): A discipline is compatible with itself.

Empty Discipline Rule: An empty discipline is compatible with all other disciplines, regardless of domain.

Domain Incompatibility Rule: Disciplines with different domain attributes are incompatible.

Potential Incompatibility Rule: Disciplines with incompatible potential natures are incompatible.

Flow Incompatibility Rule: Disciplines with incompatible flow natures are incompatible.

The following rules shall apply to determine nature compatibility:

Self Rule (Nature): A nature is compatible with itself.

Non-Existent Binding Rule: A nature is compatible with a non-existent discipline binding.

Base Nature Rule: A derived nature is compatible with its base nature.

Derived Nature Rule: Two natures are compatible if they are derived from the same base nature.

Units Value Rule: Two natures are compatible if they have the same value for the units attribute.

The following examples illustrates these rules.

Examples:

```

nature Voltage
  access = V;
  units = "V";
  abstol = 1u;
endnature

nature Current
  access = I;
  units = "A";
  abstol = 1p;
endnature

nature highvoltage: Voltage
  abstol = 1;
endnature

discipline electrical
  potential Voltage;
  nature Current;
endnature

discipline highvolt
  potential highvoltage;
  nature Current;
endnature

discipline sig_flow_v
  potential Voltage;
enddiscipline

discipline sig_flow_i
  potential Current;
enddiscipline

nature Position
  access = X;
  units = "m";
  abstol = 1u;
endnature

nature Force
  access = F;
  units = "N";
  abstol = 1n;
endnature

discipline rotational
  potential Position;
  flow Force;
enddiscipline

discipline sig_flow_x
  potential Position;
enddiscipline

discipline sig_flow_f
  potential Force;
enddiscipline

discipline empty
enddiscipline

discipline logic
  domain discrete;
enddiscipline

discipline continuous_elec
  domain continuous;
  potential Voltage;
  nature Current;
endnature

```

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

- Voltage and highvoltage are compatible natures because they both exist and are derived from the same base natures.
- electrical and highvolt are compatible disciplines because the natures for both potential and flow exist and are derived from the same base natures.
- electrical and sig_flow_v are compatible disciplines because the nature for potential is same for both disciplines and the nature for flow does not exist in sig_flow_v.

- `electrical` and `rotational` are incompatible disciplines because the natures for both potential and flow are not derived from the same base natures.
- `electrical` and `sig_flow_x` are incompatible disciplines because the nature for both potentials are not derived from the same base nature.
- An *empty discipline* is compatible with all other disciplines of the same domain (determined independently) because it does not have a potential or a flow nature. Without natures, there can be no conflicting natures.
- `electrical` and `logic` are incompatible disciplines because the domains are different. A connect statement must be used to connect nets or ports of these disciplines together.
- `electrical` and `continuous_elec` are compatible disciplines because the default domain for discipline `electrical` is `continuous` and the specified natures for potential and flow are the same.

3.9 Branches

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

Each branch declaration is associated with two nets from which it derives a discipline. These nets are referred to as the *branch terminals*. Only one net need be specified, in which case the second net defaults to `ground` and the discipline for the branch is derived from the specified net. The disciplines for the specified nets shall be compatible (see Section 3.8).

The syntax for declaring branches is shown in Syntax 3-9.

```
branch_declaration ::=
    branch list_of_branches ;
list_of_branches ::=
    terminals list_of_branch_identifiers
terminals ::=
    ( net_or_port_scalar_expression )
    | ( net_or_port_scalar_expression , net_or_port_scalar_expression )
list_of_branch_identifiers ::=
    branch_identifier [ range ]
    | branch_identifier [ range ] , list_of_branch_identifiers
```

Syntax 3-9—Syntax for branch declaration

If one of the terminals of a branch is a vector net, then the other terminal shall either be a scalar net or a vector net of the same size. In the latter case, the branch is referred to as a *vector branch*. When both terminals are vectors, the scalar branches that make up the vector branch connect to the corresponding scalar nets of the vector terminals, as shown in Figure 3-1.

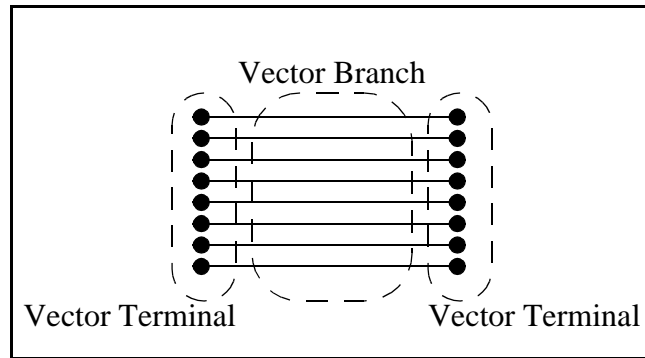


Figure 3-1 Two vector terminals

When one terminal is a vector and the other is a scalar, a singular scalar branch connects to each scalar net in the vector terminal and each terminal of the vector branch connects to the scalar terminal, as shown in Figure 3-2.

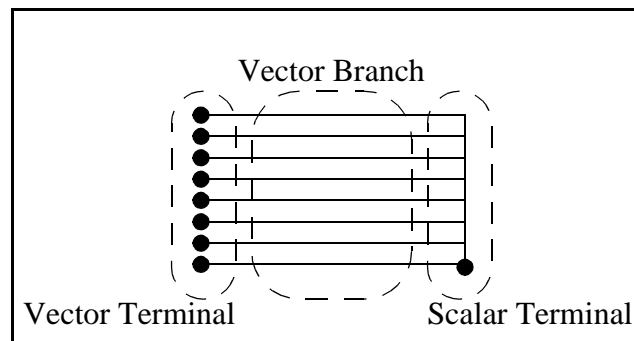


Figure 3-2 One vector and one scalar terminal

3.10 Namespace

The following subsections define the namespace.

3.10.1 Nature and discipline

Natures and disciplines are defined at the same level of scope as modules. Thus, identifiers defined as natures or disciplines have a global scope, which allows nets to be declared inside any module in the same manner as an instance of a module.

3.10.2 Access functions

Each access function name, defined before a module is parsed, is automatically added to that module's name space unless there is another identifier defined with the same name as the access function in that module's name space. Furthermore, the access function of each base nature shall be unique.

3.10.3 Net

The scope rules for net identifiers are the same as the scope rules for any other identifier declarations, except nets can not be declared anywhere other than in the port of a module or in the module itself. A net can only be declared inside a module block; a net can not be declared local to a block.

Access functions are uniquely defined for each net based on the discipline of the net. Each access function is used with the name of the net as its argument and a net can only be accessed through its access functions.

The hierarchical reference character (.) can be used to reference a net across the module boundary according to the rules specified in *IEEE 1364-1995 Verilog HDL*.

3.10.4 Branch

The scope rules for branch identifiers are the same as the scope rules for net identifiers. A branch can only be declared inside a module block; there is no local declaration for a branch.

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

The hierarchical reference character (.) can be used to reference a net across the module boundary according to the rules specified in *IEEE 1364-1995 Verilog HDL*.

Section 4

Expressions

This section describes the operators and operands available in the Verilog-AMS HDL, and how to use them to form expressions.

An *expression* is a construct which combines *operands* with *operators* to produce a result which 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 reals, without a operator is also considered an expression. Wherever a value is needed in a Verilog-AMS 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 they can use any of the operators defined in Table 4-1, Table 4-15, and Table 4-16.

4.1 Operators

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

Table 4-1—Operators

+ - * /	Arithmetic
%	Modulus
> >= < <=	Relational
!== ==	Case equality
!= ==	Logical equality
!	Logical negation
&&	Logical and
	Logical or
~	Bit-wise negation
&	Bit-wise and
	Bit-wise inclusive or
^	Bit-wise exclusive or
^~ ~^	Bit-wise equivalence

Table 4-1—Operators, *continued*

<<	Left shift
>>	Right shift
?:	Conditional
or	Event or
{ } { }	Concatenation, replication

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

unary + unary -	Unary operators
+ - * /	Arithmetic
%	Modulus
> >= < <=	Relational
== !=	Logical equality
! &&	Logical
?:	Conditional
or	Event or

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

Table 4-2 lists those operators which shall not be used to operate on real numbers.

Table 4-3—Operators not allowed for real expressions

=== !==	Case Equality
~ & ^ ^~ ~^	Bit-wise
<< >>	Shift
{ } { }	Concatenation and replication operator

4.1.1.1 Real to integer conversion

Real numbers are converted to integers 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 (0).

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.
```

4.1.1.2 Arithmetic conversion

For operands, a common data type for each operand is determined before the operator is applied. If either operand is real, the other operand is converted to real. Implicit conversion takes place when a integer number is used with a real number in an operand.

Examples:

```
a = 3 + 5.0;
// The expression "3 + 5.0" is evaluated by "casting" the
// integer 3 to the real 3.0, and the result of the expression is 8.0.


b = 1 / 2;
// The above is integer division and the result is 0.

c = 8.0 + (1/2);
// (1/2) is treated as integer division, but the result is cast to a
// real (0.0) during the addition, and the result of the
// expression is 8.0.
```

4.1.2 Binary operator precedence

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

Table 4-4—Precedence rules for binary operators

+ - ! ~ (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 associates right to left. Associativity refers to the order in which the operators having the same precedence are evaluated.

Example:

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.

Examples:

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, as long as the remaining expression does not contain analog expressions. This is called *short-circuiting* an expression evaluation.

Examples:

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

If A is known to be zero (0), the result of the expression can be determined as zero (0) without evaluating the sub-expression $B | C$.

4.1.4 Arithmetic operators

Table 4-5 shows the binary arithmetic operators.

Table 4-5—Arithmetic operators defined

$a + b$	a plus b
$a - b$	a minus b
$a * b$	a multiply by b

Table 4-5—Arithmetic operators defined, *continued*

a / b	a divide by b
$a \% b$	a modulo b

Integer division truncates any fractional part toward zero (0).

The unary arithmetic operators take precedence over the binary operators. Table 4-6 shows the unary operators.

Table 4-6—Unary operators defined

$+m$	Unary plus m (same as m)
$-m$	Unary minus m

The *modulus* operator, for example $y \% z$, gives the remainder when the first operand is divided by the second, and thus is zero (0) when z divides y exactly. The result of a modulus operation takes the sign of the first operand.

For the case of the modulus operator where either argument is real, the operation performed is

$$a \% b = a - \text{floor}(a/b) * b;$$

Table 4-7 gives examples of modulus operations.

Table 4-7—Examples of modulus operations

Modulus expression	Result	Comments
$11 \% 3$	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.
$10 \% 3.75$	2.5	$[10 - \text{floor}(10/3.75)*3.75]$ yields a remainder of 2.5.

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 zero (0) if the specified relation is *false* or the value one (1) if it is *true*.

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

Examples:

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)` is evaluated, the relational expression is evaluated first and then either zero (0) or one (1) is subtracted from `foo`. When `foo - 1 < a` is evaluated, the value of `foo` operand is reduced by one (1) and then compared with `a`.

4.1.6 Case equality operators

The *case equality operators* share the same level of precedence as the *logical equality operators*. These operators have limited support in the analog block (see Section 8.3.2). Additional information on these operators can also be found in the *IEEE 1364-1995 Verilog HDL*.

4.1.7 Logical equality operators

The *logical 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

<code>a == b</code>	a equal to b
<code>a != b</code>	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 shall be zero (0) if comparison fails, one (1) if it succeeds.

4.1.8 Logical operators

The operators *logical and* (`&&`) and *logical or* (`||`) are logical connectives. The result of the evaluation of a logical comparison can be one (1) (defined as *true*) or zero (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 zero (0) and a zero or false operand into one (1).

Examples:

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

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

However, parentheses can be used to clearly show the precedence intended, as in the following rewrite of the above example:

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

4.1.9 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 following logic tables (Table 4-10 — Table 4-14) 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

$\wedge\wedge$ $\sim\wedge$	0	1
0	1	0
1	0	1

Table 4-14—Bit-wise unary negation operator

~	
0	1
1	0

4.1.10 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 (0). The right operand is always treated as an unsigned number.

Examples:

```
integer start, result;
analog 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.

4.1.11 Conditional operator

The *conditional operator*, also known as *ternary operator*, is right associative and shall be constructed using three operands separated by two operators, as shown in Syntax 4-1.

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

Syntax 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* (any value other than zero (0)), then `expression2` is evaluated and used as the result.

4.1.12 Event or

The event **or** operator performs an or of events. See Section 6.7.2 for events and triggering of events.

4.1.13 Concatenations

A concatenation is used for joining scalar elements into compound elements (buses or arrays) for the built-in types **integer** or **real**, or for elements declared of type `net_discipline`. The concatenation shall be expressed using the brace characters (`{ }`), with commas separating the expressions within.

Examples:

```

module x;
  parameter real p1[0:2] = { 1.0, 2.0, 3.0 };
  ...
endmodule

module y;
  parameter real pole1 = 0, pole2 = 0, pole3 = 0;
  x #(.p1({pole1, pole2, pole3}) x1;
  ...
endmodule

```

Module `x` defines a real-array parameter `p1`. Module `y` instantiates `x` and overrides the array value of the parameter `p1` of module `x` via the concatenation of the scalar parameters `pole1`, `pole2`, and `pole3`.

Concatenations can be expressed using a replication multiplier.

Example:

```
{c, {2{a, b}}}
```

// equivalent to: {c, a, b, a, b}

The replication multiplier shall be a constant expression.

4.2 Built-in mathematical functions

Verilog-AMS HDL supports the following standard mathematical functions.

4.2.1 Standard mathematical functions

The standard mathematical functions supported by Verilog-AMS HDL are shown in Table 4-15. The operands shall 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.

Table 4-15—Standard functions

Function	Description	Domain
ln (<i>x</i>)	Natural logarithm	$x > 0$
log (<i>x</i>)	Decimal logarithm	$x > 0$
exp (<i>x</i>)	Exponential	$x < 80$
sqrt (<i>x</i>)	Square root	$x \geq 0$
min (<i>x</i> , <i>y</i>)	Minimum	All <i>x</i> , all <i>y</i>
max (<i>x</i> , <i>y</i>)	Maximum	All <i>x</i> , all <i>y</i>
abs (<i>x</i>)	Absolute	All <i>x</i>

Table 4-15—Standard functions, *continued*

Function	Description	Domain
pow (<i>x</i> , <i>y</i>)	Power (x^y)	if $x \geq 0$, all <i>y</i> ; if $x < 0$, int (<i>y</i>)
floor (<i>x</i>)	Floor	All <i>x</i>
ceil (<i>x</i>)	Ceiling	All <i>x</i>

The **min**(), **max**(), and **abs**() functions have discontinuous derivatives; it is necessary to define the behavior of the derivative of these functions at the point of the discontinuity. In this context, these functions are defined so:

min(*x*,*y*) is equivalent to $(x < y) ? x : y$

max(*x*,*y*) is equivalent to $(x > y) ? x : y$

abs(*x*) is equivalent to $(x > 0) ? x : -x$

4.2.2 Transcendental functions

The trigonometric and hyperbolic functions supported by Verilog-AMS HDL are shown in Table 4-16. All operands shall be numeric (**integer** or **real**) and are converted to real if necessary.

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

Table 4-16—Trigonometric and hyperbolic functions

Function	Description	Domain
sin (<i>x</i>)	Sine	All <i>x</i>
cos (<i>x</i>)	Cosine	All <i>x</i>
tan (<i>x</i>)	Tangent	$x \neq n(\pi/2)$, <i>n</i> is odd
asin (<i>x</i>)	Arc-sine	$-1 \leq x \leq 1$
acos (<i>x</i>)	Arc-cosine	$-1 \leq x \leq 1$
atan (<i>x</i>)	Arc-tangent	All <i>x</i>
atan2 (<i>x</i> , <i>y</i>)	Arc-tangent of <i>x</i> / <i>y</i>	All <i>x</i> , All <i>y</i>
hypot (<i>x</i> , <i>y</i>)	sqrt ($x^2 + y^2$)	All <i>x</i> , All <i>y</i>
sinh (<i>x</i>)	Hyperbolic sine	$x < 80$
cosh (<i>x</i>)	Hyperbolic cosine	$x < 80$
tanh (<i>x</i>)	Hyperbolic tangent	All <i>x</i>
asinh (<i>x</i>)	Arc-hyperbolic sine	All <i>x</i>
acosh (<i>x</i>)	Arc-hyperbolic cosine	$x \geq 1$
atanh (<i>x</i>)	Arc-hyperbolic tangent	$-1 \leq x \leq 1$

4.2.3 Error handling

All values outside of the domain for these math functions shall report an error.

4.3 Signal access functions

Access functions are used to access signals on nets, ports, and branches. There are two types of access functions, *branch access functions* and *port access functions*. The name of the access function for a signal is taken from the discipline of the net, port, or branch where the signal or port is associated and utilizes the () operator. A port access function also takes its name from the discipline of the port to which it is associated but utilizes the port access (<>) operator.

If the signal or port access function is used in an expression, the access function returns the value of the signal. If the signal access function is being used on the left side of a branch assignment or contribution statement, it assigns a value to the signal. A port access function can not be used on the left side of a branch assignment or contribution statement.

Table 4-17 shows how access functions can be applied to branches, nets, and ports. In this table, *b1* refers to a branch, *n1* and *n2* represent either nets or ports, and *p1* represents a port. These branches, nets, 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).

Table 4-17—Access functions examples

Example	Comments
$V(b1)$	Accesses the voltage across branch <i>b1</i> .
$V(n1)$	Accesses the voltage of <i>n1</i> (a net or a port) relative to ground.
$V(n1, n2)$	Accesses the voltage difference between <i>n1</i> and <i>n2</i> (nets or ports).
$I(b1)$	Accesses the current on branch <i>b1</i> .
$I(n1)$	Accesses the current flowing from <i>n1</i> (a net or port) to ground.
$I(n1, n2)$	Accesses the current flowing between <i>n1</i> and <i>n2</i> .
$I(n1, n1)$	Error
$I(<p1>)$	Accesses the current flow into the module through port <i>p1</i> .

The argument expression list for signal access functions shall be a branch identifier, or a list of one or two nets or port expressions. If two net expressions are given as arguments to a flow access function, they shall not evaluate to the same signal. The net identifiers shall be scalar or resolve to a constant net of a composite net type (array or bus) accessed by a genvar expression.

The operands of an expression shall be unique to define a valid branch. The access function name shall match the discipline declaration for the nets, ports, or branch given in the argument expression list. In this case, v and i are used as examples of access functions for electrical potential and flow.

For port access functions, the expression list is a single port of the module. The port identifier shall be scalar or resolve to a constant net of a bus port accessed by a genvar expression. The access function name shall match the discipline declaration for the port identifier.

4.4 Analog operators

Analog operators operate on an expression and return a value. They are functions which operate on more than just the current value of their arguments. Instead, they maintain their internal state and their output is a function of both the input and the internal state.

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, which 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 to improve convergence.

4.4.1 Restrictions on analog operators

Analog operators are subject to several important restrictions because they maintain their internal state.

- Analog operators shall not be used inside conditional (**if** and **case**) or looping (**for**) statements unless the conditional expression controlling the statement consists of terms which can not change their value during the course of an analysis, i.e., unless the conditional expression is a genvar expression.
- Analog operators are not allowed in the **repeat** and **while** looping statements.
- Analog operators can only be used inside an **analog** block; they can not be used inside an **initial** or **always** block, or inside a user-defined function.
- It is illegal to specify a null argument in the argument list of an analog operator, except as specified elsewhere in this document.

These restrictions help prevent usage which could cause the internal state to be corrupted or become out-of-date, which results in anomalous behavior.

4.4.2 Vector or array arguments to analog operators

Certain analog operators require arrays or vectors to be passed as arguments: Laplace filters, Z-transform filters, and `noise_table`. An array can be passed as:

- *array_identifier*
- *const_array_expression*

A *const_array_expression* allows the arrays to be passed within the argument list without explicit declaration of the array object, as shown in Syntax 4-2.

```
constant_array_expression ::=
    { constant_arrayinit_element { , constant_arrayinit_element } }

constant_arrayinit_element ::=
    constant_expression
    | integer_constant_expression { constant_expression }
```

Syntax 4-2—Syntax for constant array expression

4.4.3 Analog operators and equations

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, some criteria (*tolerances*) is needed to determine when the algorithm knows when it is close enough to the solution and then stops the iteration. Thus, each equation, at a minimum, shall have a tolerance defined and associated with it.

Occasionally, analog operators require 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 attempts to determine from context which tolerance to associate with the new equation and new unknown. Alternatively, these operators can be used to specify tolerances.

Specifying natures also directly enforces reusability and allows other signal attributes to be accessed by the simulator.

4.4.4 Time derivative operator

The `ddt` operator computes the time derivative of its argument, as shown in Table 4-18.

Table 4-18—Time derivative

Operator	Comments
ddt (<i>expr</i>)	Returns $\frac{d}{dt}x(t)$, the time-derivative of <i>x</i> , where <i>x</i> is <i>expr</i> .
ddt (<i>expr</i> , <i>abstol</i>)	Same as above, except absolute tolerance is specified explicitly.
ddt (<i>expr</i> , <i>nature</i>)	Same as above, except nature is specified explicitly.

In DC analysis, **ddt**() returns zero (0). The optional parameter *abstol* is used as an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context where **ddt**() is used. See Section 4.4.3 for more information on the application of tolerances to equations. The absolute tolerance, *abstol* or derived from *nature*, applies to the output of the **ddt** operator and is the largest signal level that is considered negligible.

4.4.5 Time integral operator

The **idt** operator computes the time-integral of its argument, as shown in Table 4-19.

Table 4-19—Time integral

Operator	Comments
idt (<i>expr</i>)	Returns $\int_0^t x(\tau) d\tau$, the time-integral of <i>x</i> from 0 to <i>t</i> with the initial condition being computed in the DC analysis. Where <i>x</i> is <i>expr</i> .
idt (<i>expr</i> , <i>ic</i>)	Returns $\int_0^t x(\tau) d\tau + ic$, the time-integral of <i>x</i> from 0 to <i>t</i> with the initial condition <i>ic</i> . In DC analysis, <i>ic</i> is returned. Where <i>x</i> is <i>expr</i> .
idt (<i>expr</i> , <i>ic</i> , <i>assert</i>)	Returns $\int_{t_0}^t x(\tau) d\tau + ic$, the time-integral of <i>x</i> from <i>t</i> ₀ to <i>t</i> with initial condition <i>ic</i> . <i>assert</i> is a integer-valued expression. idt returns <i>ic</i> when <i>assert</i> is non-zero. <i>t</i> ₀ is the time when <i>assert</i> last became 0. Where <i>x</i> is <i>expr</i> .
idt (<i>expr</i> , <i>ic</i> , <i>assert</i> , <i>abstol</i>)	Same as above, except absolute tolerance is specified explicitly.
idt (<i>expr</i> , <i>ic</i> , <i>assert</i> , <i>nature</i>)	Same as above, except nature is specified explicitly.

When specified with initial conditions, **idt**() returns the value of the initial condition in DC and IC analyses whenever *assert* is given and is non-zero. Without initial conditions, **idt**() multiplies its argument by infinity in DC analysis. Hence, without initial conditions, it can only be used in a system with feedback which forces its argument to zero (0).

The optional parameter *abstol* or *nature* is used to derive an absolute tolerance if needed. Whether an absolute tolerance is needed depends on the context where **idt**() is used. (See Section 4.4.3 for more information.) The absolute tolerance applies to the input of the **idt** operator and is the largest signal level that is considered negligible.

4.4.6 Circular integrator operator

The **idtmod** operator, also called the *circular integrator*, converts an expression argument into its indefinitely integrated form similar to the **idt** operator, as shown in Table 4-20.

Table 4-20—Circular integrator

Operator	Comments
idtmod (<i>expr</i>)	Returns $\int_0^t x(\tau) d\tau$, the time-integral of <i>x</i> from 0 to <i>t</i> with the initial condition being computed in the DC analysis. Where <i>x</i> is <i>expr</i> .
idtmod (<i>expr</i> , <i>ic</i>)	Returns $\int_0^t x(\tau) d\tau + ic$, the time-integral of <i>x</i> from 0 to <i>t</i> with the initial condition <i>ic</i> . In DC analysis, <i>ic</i> is returned. Where <i>x</i> is <i>expr</i> .
idtmod (<i>expr</i> , <i>ic</i> , <i>modulus</i>)	Returns <i>k</i> , where $0 \leq k < \text{modulus}$ and <i>k</i> is $\int_0^t x(\tau) d\tau + ic = n \times \text{modulus} + k$, $n = \dots -3, -2, -1, 0, 1, 2, 3, \dots$. Where <i>x</i> is <i>expr</i> .
idtmod (<i>expr</i> , <i>ic</i> , <i>modulus</i> , <i>offset</i>)	Returns <i>k</i> , where $\text{offset} \leq k < \text{offset} + \text{modulus}$ and <i>k</i> is $\int_0^t x(\tau) d\tau + ic = n \times \text{modulus} + k$. Where <i>x</i> is <i>expr</i> .
idtmod (<i>expr</i> , <i>ic</i> , <i>modulus</i> , <i>offset</i> , <i>abstol</i>)	Same as above, except absolute tolerance is specified explicitly.
idtmod (<i>expr</i> , <i>ic</i> , <i>modulus</i> , <i>offset</i> , <i>nature</i>)	Same as above, except nature is specified explicitly.

The initial condition is optional. If the initial condition is not specified, it defaults to zero (0). Regardless, the initial condition shall force the DC solution to the system.

If **idtmod**() is used in a system with feedback configuration which forces *expr* to zero (0), the initial condition can be omitted without any unexpected behavior during simulation. For example, an operational amplifier alone needs an initial condition, but the same amplifier with the right external feedback circuitry does not need a forced DC solution.

The output of the **idtmod**() function shall remain in the range

$$\text{offset} \leq \text{idtmod} < \text{offset} + \text{modulus}$$

The *modulus* shall be an expression which evaluates to a positive value. If the *modulus* is not specified, then **idtmod**() shall behave like **idt**() and not limit the output of the integrator.

The default for *offset* shall be zero (0).

The following relationship between **idt**() and **idtmod**() shall hold at all times.

Examples:

If

```
y = idt(expr, ic) ;
z = idtmod(expr, ic, modulus, offset) ;
```

then

```
y = n * modulus + z ;// n is an integer
```

where

```
offset ≤ z < modulus + offset
```

In this example, the circular integrator is useful in cases where the integral can get very large, such as a VCO. In a VCO we are only interested in the output values in the range $[0, 2\pi]$, e.g.,

```
phase = idtmod(fc + gain*V(in), 0, 1, 0);
V(OUT) <+ sin(2*'M_PI'*phase);
```

Here, the circular integrator returns a value in the range $[0, 1]$.

4.4.7 Absolute delay operator

absdelay() implements the absolute transport delay for continuous waveforms (use the **transition** operator to delay discrete-valued waveforms). The general form is

```
absdelay( input, td [, maxdelay ] )
```

input is delayed by the amount *td*. In all cases *td* shall be a positive number. If the optional *maxdelay* is specified, then *td* can vary; but it shall be an error if it becomes larger than *maxdelay*. If *maxdelay* is not specified, changes to *td* shall be ignored. If *maxdelay* is specified, changes to *td* are ignored and the initial value of *maxdelay* is used.

In DC and operating point analyses, **absdelay()** returns the value of its *input*. In AC and other small-signal analyses, the **absdelay** operator phase-shifts the input expression to the output of the delay operator based on the following formula.

$$Output(\omega) = Input(\omega) \cdot e^{-j\omega td}$$

td is evaluated as a constant at a particular time for any small signal analysis. In time-domain analyses, **absdelay()** introduces a transport delay equal to the instantaneous value of *td* based on the following formula.

$$Output(t) = Input(\max(t - td, 0))$$

The transport delay *td* can be either constant (typical case) or vary as a function of time (when *maxdelay* is defined). A time-dependent transport delay is shown in Figure 4-1, with a ramp input to the **absdelay** operator for both positive and negative changes in the transport delay *td* and a *maxdelay* of 5.

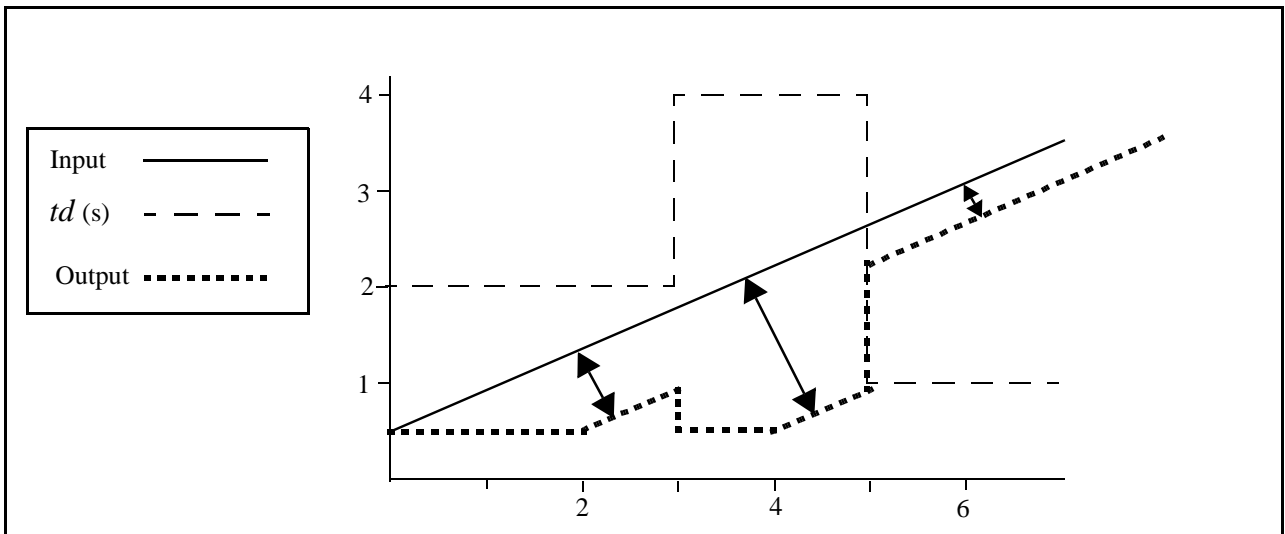


Figure 4-1 Transport delay example

From time 0 until 2s, the output remains at `input(0)`. With a delay of 2s, the output then starts tracking `input(t - 2)`. At 3s, the transport delay changes from 2s to 4s, switching the output back to `input(0)`, since `input(max(t-td,0))` returns 0. The output remains at this level until 4s when it once again starts tracking the input from $t = 0$. At 5s the transport delay goes to 1s and the output correspondingly jumps from its current value to the value defined by `input(t - 1)`.

4.4.8 Transition filter

transition() smooths out piecewise constant waveforms. The transition filter is used to imitate transitions and delays on digital signals (for non-piecewise constant signals, see Section 4.4.9). This function provides controlled transitions between discrete signal levels by setting the rise time and fall time of signal transitions, as shown in Figure 4-2.

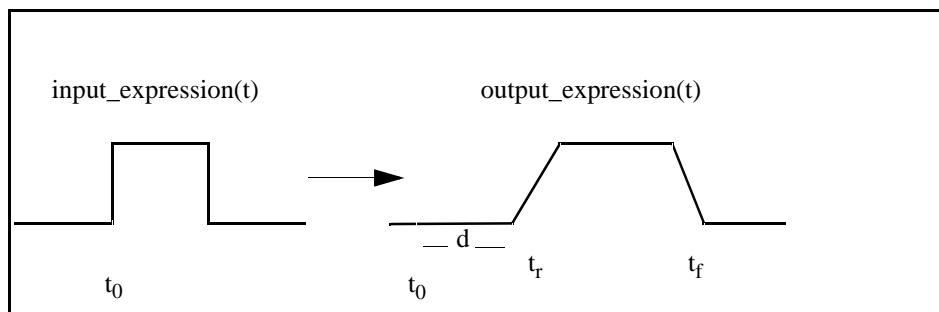


Figure 4-2 Transition filter example

transition() stretches instantaneous changes in signals over a finite amount of time and can delay the transitions, as shown in Figure 4-3.

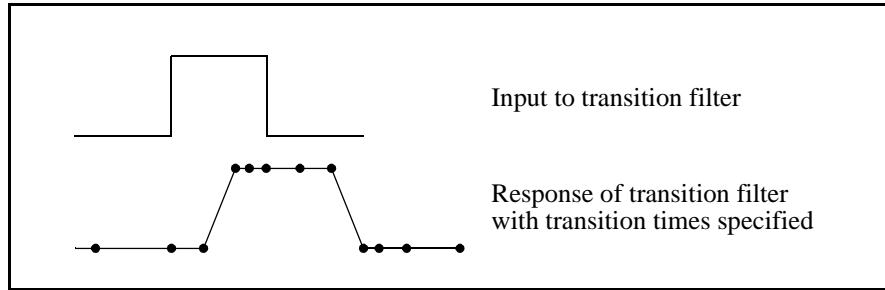


Figure 4-3 Shifting the transition filter

4.4.8.1 Syntax

The general form is

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

transition() takes the following arguments (all real-valued expressions):

- the input expression
- the delay time (shall be non-negative)
- the rise time (shall be greater than or equal to 0)
- the fall time (shall be greater than or equal to 0)
- the time_tol (shall be positive)

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

transition() returns a *real* number which describes a piecewise linear function over time. The transition function causes the simulator to place time-points at both corners of a transition. If *time_tol* is not specified, the transition function causes the simulator to assure each transition is adequately resolved.

td, *rise_time*, *fall_time*, and *time_tol* are optional. If *td* is not specified, it is taken to be zero (0.0). If only a positive *rise_time* value is specified, the simulator uses it for both rise and fall times. If neither rise nor fall time are specified or are equal to zero (0.0), the rise and fall time default to the value defined by `'default_transition`.

If `'default_transition` is not specified the default behavior approximates the ideal behavior of a zero-duration transition. Forcing a zero-duration transition is undesirable because it could cause convergence problems. Instead, a negligible, but non-zero, transition time is used. The small non-zero transition time allows the simulator to shrink the timestep small enough so a smooth transition occurs and any convergence problems are avoided. The simulator does not force a time point at the trailing corner of a transition

to avoid causing the simulator to take very small time steps, which would result in poor performance.

In DC analysis, **transition()** passes the value of the *expr* directly to its output. The **transition** filter is designed to smooth out piecewise constant waveforms. When applied to waveforms which 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()** (see Section 4.4.9) 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.

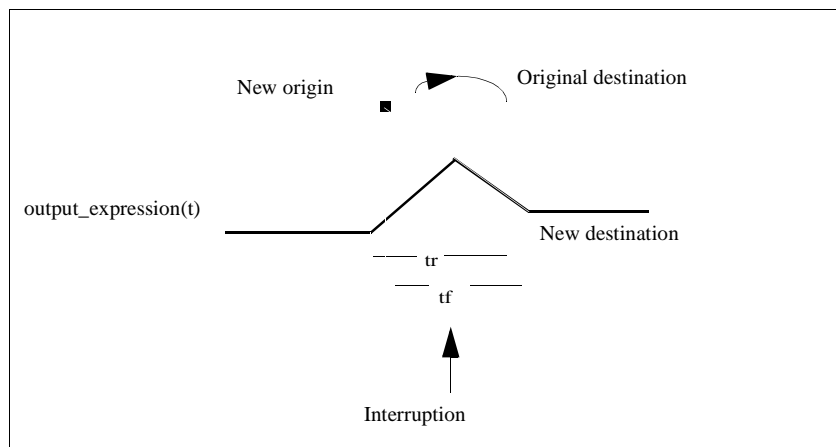


Figure 4-4 Completing the transition

Figure 4-4 shows a rising transition is interrupted near its midpoint and the new destination level of the value below the current value. For the new origin and destination, **transition()** computes the slope which 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 different 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 which 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 can not 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.

4.4.8.2 Examples

Example 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, dly=0, ttime=1.0/freq ;
input [0:4] in ;
output out ;
electrical [0:4] in;
electrical out ;
real x, y, thresh;
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, dly, ttime) ;
    y = transition(col - 1.5, dly, ttime) ;
    V(out) <+ ampl*x*cos(2*'M_PI'*freq*$abstime)
        + ampl*y*sin(2*'M_PI'*freq*$abstime) ;
    $bound_step(0.1/freq) ;
end
endmodule

```

Example 2 - A/D converter

In this example, an analog behavioral N-bit analog to digital converter, demonstrates the ability of the transition function to handle vectors.

```

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

analog begin
    @(cross(V(clk)-2.5, +1)) begin
        sample = V(in);
        thresh = fullscale/2.0;
        for (i = bits - 1; i >= 0; i = i - 1) begin
            if (sample > thresh)

```

```

        begin
        result[i] = 1.0;
        sample = sample - thresh ;
        end else begin
        result[i] = 0.0;
        end
        sample = 2.0*sample;
    end
end
for (i = 0; i < bits; i = i + 1) begin
    V(out[i]) <+ transition(result[i], dly, ttime);
end
end
endmodule

```

4.4.9 Slew filter

The **slew** analog operator 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 Section 4.4.8.) The general form is

slew(*expr* [, *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 as shown in Figure 4-5.

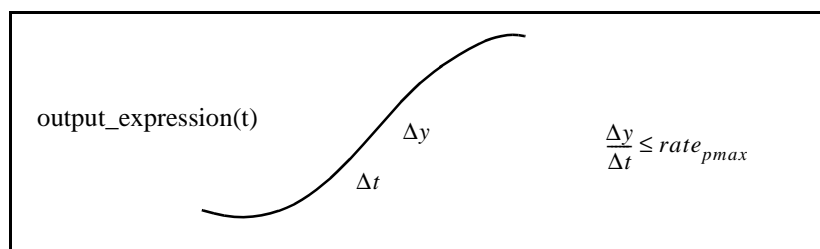


Figure 4-5 Slew filter transition

The two rate values are optional. *max_pos_slew_rate* shall be greater than zero (0) and *max_neg_slew_rate* shall be less than zero (0). If the *max_neg_slew_rate* is not specified, it defaults to the inverse of the *max_pos_slew_rate*. If no rates are specified, **slew()** passes the signal through unchanged. If the rate of change of *expr* is less than the specified maximum slew rates, **slew()** returns the value of *expr*.

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.

4.4.10 last_crossing function

Related to the **cross()** function, the **last_crossing()** function returns a real value representing the simulation time when a signal expression last crossed zero (0). The general form is

```
last_crossing( expr , direction ) ;
```

The *direction* flag is interpreted in the same way as in 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; it uses linear interpolation to estimate the time of the last crossing. However, it can be used with the **cross** function for improved accuracy.

Examples:

The following example measures the period of its input signal using the **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 (0) for the first time, the **last_crossing()** function returns a negative value.

4.4.11 Laplace transform filters

The Laplace transform filters implement lumped linear continuous-time filters. Each filter takes an optional parameter ϵ , which is a real number or a nature used for deriving an absolute tolerance (if needed). Whether an absolute tolerance is needed depends on the context where the filter is used. The zeros argument may be represented as a null argument. The null argument is characterized by two adjacent commas (, ,) in the argument list.

4.4.11.1 `laplace_zp`

`laplace_zp()` implements the zero-pole form of the Laplace transform filter. The general form is

`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 k^{th} zero (0), while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a root (a pole or zero) is real, the imaginary part shall be specified as zero (0). If a root is complex, its conjugate shall 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).

4.4.11.2 `laplace_zd`

`laplace_zd()` implements the zero-denominator form of the Laplace transform filter. The general form is

`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 containing the coefficients of the denominator. The 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 k^{th} zero, while d_k is the coefficient of the k^{th} power of s in the denominator. If a zero is real, the imaginary part shall be specified as zero (0). If a zero is complex, its conjugate shall also be present. If a zero is zero (0), then the term associated with it is implemented as s , rather than $(1 - s/\zeta)$.

4.4.11.3 `laplace_np`

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

`laplace_np(expr , n , ρ [, ε])`

where n is a vector of M real numbers containing 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 k^{th} power of s in the numerator, while ρ_k^r and ρ_k^i are the real and imaginary parts of the k^{th} pole. If a pole is real, the imaginary part shall be specified as zero (0). If a pole is complex, its conjugate shall also be present. If a pole is zero (0), then the term associated with it is implemented as s , rather than $(1 - s/\rho)$.

4.4.11.4 `laplace_nd`

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

The general form is

laplace_nd(expr , n , d [, ε])

where n is an vector of M real numbers containing the coefficients of the numerator and d is a vector of N real numbers containing 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 k^{th} power of s in the numerator and d_k is the coefficient of the k^{th} power of s in the denominator.

4.4.11.5 Examples

V(out) <+ **laplace_zp**(V(in) , { -1,0 } , { -1,-1,-1,1 });

implements

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

and

V(out) <+ **laplace_nd**(V(in) , { 0,1 } , { -1,0,1 });

implements

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

This example

V(out) <+ **laplace_zp**(white_noise(k) , , { 1,0,1,0,-1,0,-1,0 });

implements a band-limited white noise source as

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

4.4.12 Z-transform filters

The *Z-transform* filters implement linear discrete-time filters. Each filter supports a parameter T which specifies the sampling period of the filter. A filter with unity transfer function acts like a simple sample-and-hold which samples every T seconds and exhibits

no delay. The zeros argument may be represented as a null argument. The null argument is characterized by two adjacent commas (, ,) in the argument list.

All Z-transform filters share three common arguments: T , τ , and t_0 . T specifies the period of the filter, is mandatory, and shall be positive. τ specifies the transition time, is optional, and shall be nonnegative.

If the transition time is specified and is non-zero, 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 (1) unit of time (as defined by the ``default_transition` compiler directive) and the timestep is not controlled to resolve the trailing corner of the transition. If the transition time is specified as zero (0), then the output is abruptly discontinuous. A Z-filter with zero (0) transition time shall not 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 $t=0$.

4.4.12.1 `zi_zp`

`zi_zp()` implements the zero-pole form of the Z-transform filter. The general form is

`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 (0) 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

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

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

4.4.12.2 `zi_zd`

`zi_zd()` implements the zero-denominator form of the Z-transform filter.

The form is

zi_zd(expr , ζ , d , $T[, \tau[, t_0]]$)

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 containing the coefficients of the denominator. The transfer function is

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

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

4.4.12.3 **zi_np**

zi_np() implements the numerator-pole form of the Z-transform filter. The general form is

zi_np(expr , n , ρ , $T[, \tau[, t_0]]$)

where n is a vector of M real numbers containing 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

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

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

4.4.12.4 **zi_nd**

zi_nd() implements the numerator-denominator form of the Z-transform filter. The general form is

zi_nd(expr , n , d , T [, τ [, t₀]])

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

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

where n_k is the coefficient of the k^{th} power of s in the numerator and d_k is the coefficient of the k^{th} power of s in the denominator.

4.4.13 **Limited exponential**

The **limexp()** function is an operator whose internal state contains information about the argument on previous iterations. It returns a real value which 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 general form is

limexp (expr)

The apparent behavior of **limexp()** is not distinguishable from **exp()**, except using **limexp()** to model semiconductor junctions generally results in dramatically improved convergence. There are different ways of implementing limiting algorithms for the exponential^{1 2}.

1. Laurence W. Nagel, "SPICE2: A computer program to simulate semiconductor circuits," Memorandum No. ERL-M520, University of California, Berkeley, California, May 1975.

2. W. J. McCalla, *Fundamentals of Computer-Aided Circuit Simulation*. Kluwer Academic Publishers, 1988.

4.4.14 Constant versus dynamic arguments

Some of the arguments to the analog operators described in this section and the events described in Section 6 expect dynamic expressions and others expect constant expressions. The dynamic expressions can be functions of circuit quantities and can change during an analysis. The constant expressions remain static throughout an analysis.

Table 4-21 summarizes the arguments of the analog operators defined in this section.

Table 4-21—Analog operator arguments

Operator	Constant expression arguments	Dynamic expression arguments
ddt	abstol	expr
idt	abstol	expr, ic, assert
idtmod	abstol	expr, ic, modulus, offset
cross	expr_tol, time_tol	expr, dir
last_crossing		expr, dir
absdelay	maxdelay	expr, td
transition		expr, td, rise_time, fall_time
slew		expr, max_pos_slew_rate, max_neg_slew_rate
zi_zp zi_zd zi_np zi_nd	zeros, poles, T , t_0	expr, τ
laplace_zp laplace_zd laplace_np laplace_nd	poles, abstol, zero	expr
timer	time_tol	start_time, period
limexp		expr

If a dynamic expression is passed as an argument which expects a constant expression, the value of the dynamic expression at the start of the analysis defaults to the constant value of the argument. Any further change in value of that expression is ignored during the iterative analysis.

4.5 Analysis dependent functions

This section describes the **analysis()** function, which is used to determine what type of analysis is being performed, and the small-signal source functions. 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 zero (0).

4.5.1 Analysis

The **analysis()** function takes one or more string arguments and returns one (1) if any argument matches the current analysis type. Otherwise it returns zero (0). The general form is

analysis(analysis_list)

There is no fixed set of analysis types. Each simulator can support its own set. However, simulators shall use the types listed in Table 4-22 to represent analyses which are similar to those provided by *SPICE*.

Table 4-22—Analysis types

Name	Analysis description
"ac"	.AC analysis
"dc"	.OP or .DC analysis
"noise"	.NOISE analysis
"tran"	.TRAN analysis
"ic"	The initial-condition analysis which precedes 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 which precedes an AC or noise analysis, or the IC analysis which precedes a transient analysis.
"nodeset"	The phase during an equilibrium point calculation where nodesets are forced.

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

Table 4-23 describes the implementation of the analysis function. Each column shows the return value of the function. A status of one (1) represents *True* and zero (0) represents *False*.

Table 4-23—Return values for analysis functions

Analysis	Argument	Simulator analysis type					
		DC	TRAN	AC	NOISE	DC	NOISE
		OP	TRAN	OP	AC	OP	AC
First part of "static"	"nodeset"	1	1	0	1	0	1
Initial DC state	"static"	1	1	0	1	0	1
Initial condition	"ic"	0	1	0	0	0	0
DC	"dc"	1	0	0	0	0	0
Transient	"tran"	0	1	1	0	0	0

Table 4-23—Return values for analysis functions, *continued*

Analysis	Argument	Simulator analysis type							
		DC	TRAN		AC		NOISE		
		OP	TRAN	OP	AC	OP	AC	OP	AC
Small-signal	"ac"	0	0	0	1	1	0	0	
Noise	"noise"	0	0	0	0	0	1	1	

Using the `analysis()` function, it is possible to have a module behave differently depending on which analysis is being run.

Examples:

To implement nodesets or initial conditions using the analysis function and switch branches, use the following.

```
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 which has been linearized about its operating point and is driven by a small sinusoid. The sinusoidal stimulus is provided using the `ac_stim()` function. The general form is

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

The AC stimulus function returns zero (0) during large-signal analyses (such as DC and transient) as well as on all small-signal analyses using names which do not match *analysis_name*. The name of a small-signal analysis is implementation dependent, although the expected name (of the equivalent of a *SPICE* AC analysis) is "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 one (1) and the default phase is zero (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()` system task. 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 zero (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 their spectral density does not depend on frequency. They are modeled using

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

which generates white noise with a power of *pwr*.

Examples:

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 used when 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 **flicker_noise()** function models flicker noise. The general form is

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

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

The optional *name* argument acts as a label for the noise source used when 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 **noise_table()** function interpolates a vector to model a process where the spectral density of the noise varies as a piecewise linear function of frequency. The general form is

```
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 pairs are specified in the order of ascending frequencies. **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 used when 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 shown in the following example.

```
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 which 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.

Examples:

Example 1 - Two noise voltages are perfectly correlated.

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

Example 2 - Partially correlated noise sources can also be modelled.

```
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

A user-defined function can be used to return a value (for an expression). All functions are defined within modules. Each function can be an analog function or a digital function (as defined in *IEEE 1364-1995 Verilog HDL*).

4.6.1 Defining an analog function

The syntax for defining an analog function is shown in Syntax 4-3.

```

analog_function_declaration ::=
    analog 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

```

Syntax 4-3—Syntax for an analog function declaration

An analog function declaration shall begin with the keywords **analog function**, optionally followed by the type of the return value from the function, then the name of the function and a semicolon, and ending with the keyword **endfunction**.

type specifies the return value of the function; its use is optional. *type* can be a `real` or an `integer`; if unspecified, the default is `real`.

An analog function:

- can use any statements available for conditional execution (see Section 6.1);
- shall not use access functions;
- shall not use contribution statements or event control statements;
- shall have at least one input declared; the block item declaration shall declare the type of the inputs as well as local variables used in the function.
- shall not use named blocks; and
- shall only reference locally-defined variables or variables passed as arguments.

Examples:

The following example defines an analog function called `maxValue`, which returns the potential of whichever signal is larger.

```

analog function real maxValue;
input n1, n2 ;
real n1, n2 ;
begin
    // code to compare potential of two signal
    maxValue = (n1 > n2) ? n1 : n2 ;
end
endfunction

```

4.6.2 Returning a value from an analog function

The analog function definition implicitly declares a variable, internal to the analog function, with the same name as the analog function. This variable has the same type as the type specified in the analog function declaration. The analog function definition initializes the return value from the analog function by assigning the analog function result to the internal variable with the same name as the analog function. This variable can be read and assigned within the flow; its last assigned value is passed back on the return call.

Example:

The following line (from the example in Section 4.6.1) illustrates this concept:

```
maxValue = (n1 > n2) ? n1 : n2 ;
```

If the internal variable is not assigned, the function shall return zero (0).

4.6.3 Calling an analog function

An analog function call is an operand within an expression. Syntax 4-4 shows the analog function call.

```

analog_function_call ::=
    function_identifier ( expression { , expression } )

```

Syntax 4-4—Syntax for function call

The order of evaluation of the arguments to an analog function call is undefined.

An analog function:

- shall not call itself directly or indirectly, i.e., recursive functions are not permitted;
- shall only be called within an analog block; and
- can be called outside of their immediate scope.

Example:

The following example uses the `maxValue` function defined in Section 4.6.1.

```
V(out) <+ maxValue(val1, val2) ;
```

Section 5

Signals

5.1 Analog signals

Analog signals are distinguished from digital signals in that an *analog signal* has a discipline with a continuous domain. Disciplines, nets, nodes, and branches are described in Section 3 and ports are described in Section 7.

This section describes analog branch assignments, signal access mechanisms, and operators in Verilog-AMS HDL.

5.1.1 Access functions

Flows and potentials on nets, ports, and branches are accessed using *access functions*. The name of the access function is taken from the discipline of the net, port, or branch associated with the signal.

Examples:

Example 1

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) is accessed via $v(b)$ and the flow (current) is accessed via $i(b)$.

- There can be any number of named branches between any two signals.
- Unnamed branches are accessed in a similar manner, except the access functions are applied to net names or port names rather than branch names.

Example 2

If *n1* and *n2* are electrical nets or ports, then $v(n1, n2)$ creates an unnamed branch from *n1* to *n2* (if it does not already exist) and then accesses the branch potential (or the potential difference between *n1* to *n2*), and $v(n1)$ does the same from *n1* to the global reference node (*ground*).

- In other words, accessing the potential from a net or port to a net or port defines an unnamed branch. Accessing the potential on a single net or port defines an unnamed branch from that net or port to the global reference node (*ground*). There can only be one unnamed branch between any two nets.
- An analogous access method is used for flows.

Example 3

$I(n1, n2)$ creates an unnamed branch from $n1$ to $n2$ (if it does not already exist) and then accesses the branch flow, and $I(n1)$ does the same from $n1$ to the global reference node (*ground*).

- Thus, accessing the flow from a net or port to a net or port defines an unnamed branch. Accessing the potential on a single net or port defines an unnamed branch from that net or port to the global reference node (*ground*).
- It is also possible to access the flow passing through a port into a module. The name of the access function is derived from the flow nature of the discipline of the port. In this case, ($\langle \rangle$) is used to delimit the port name rather than (\cdot).

Example 4

$I(\langle p1 \rangle)$ is used to access the current flow into the module through the electrical port $p1$. This capability is discussed further in Section 5.1.4.

5.1.2 Probes and sources

An analog component can be represented using a network of probes and controlled sources. The Verilog-AMS HDL uses the concept of *probes* and *sources* as a means of unambiguously representing a network. The mapping between these representations are defined in following subsections.

5.1.2.1 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 illegal. The models for probe branches are shown in Figure 5-1.

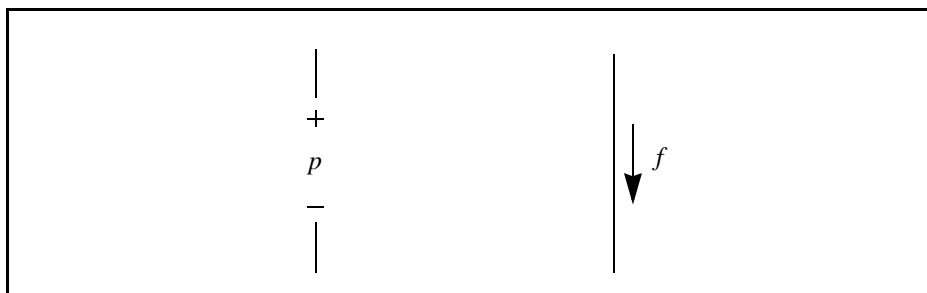


Figure 5-1 Equivalent circuit models for probe branches

The branch potential of a flow probe is zero (0). The branch flow of a potential probe is zero (0).

5.1.2.2 Sources

A branch, either named or unnamed, is a *source branch* if either the potential or the flow of that branch is assigned a value by a contribution statement (see Section 5.3) anywhere in the module. It is a *potential source* if the branch potential is specified and is a *flow source* if the branch flow is specified. A branch cannot simultaneously be both a potential and a flow source, although it can switch between them (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 in Figure 5-2.

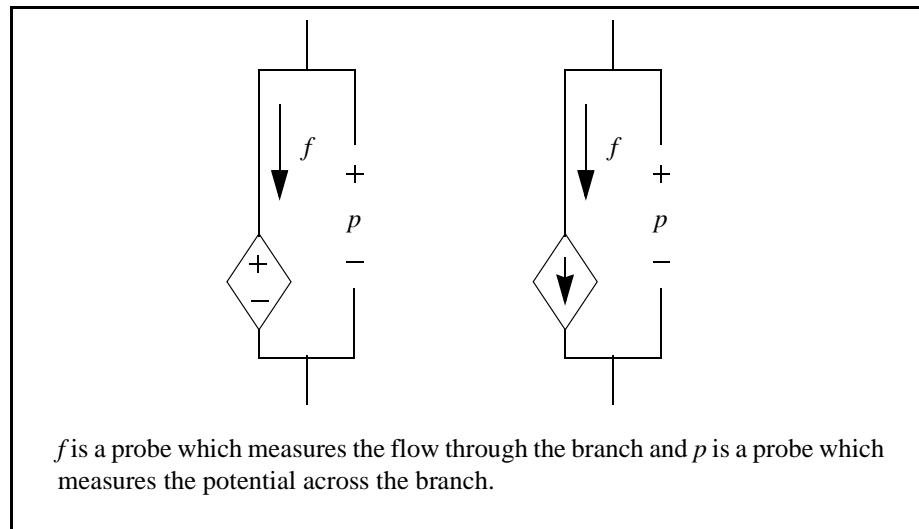


Figure 5-2 Equivalent circuit models for source branches

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.

5.1.3.1 The four controlled sources

The following example is used with each of the four behavioral statements listed below. Each statement creates a unique controlled source when inserted into this example.

```

module control_source (p, n, ps, ns);
  electrical p, n, ps, ns;
  parameter A=1;
  branch (ps,ns) in;
  branch (p,n) out;
  analog begin
    // add behavioral statement here
  end
endmodule

```

The model for a voltage controlled voltage source is

$$V(\text{out}) <+ A * V(\text{in});$$

The model for a voltage controlled current source is

$$I(\text{out}) <+ A * V(\text{in});$$

The model for a current controlled voltage source is

$$V(\text{out}) <+ A * I(\text{in});$$

The model for a current controlled current source is

$$I(\text{out}) <+ A * I(\text{in});$$

5.1.3.2 Resistor and conductor

Figure 5-3 shows the model for a linear conductor.

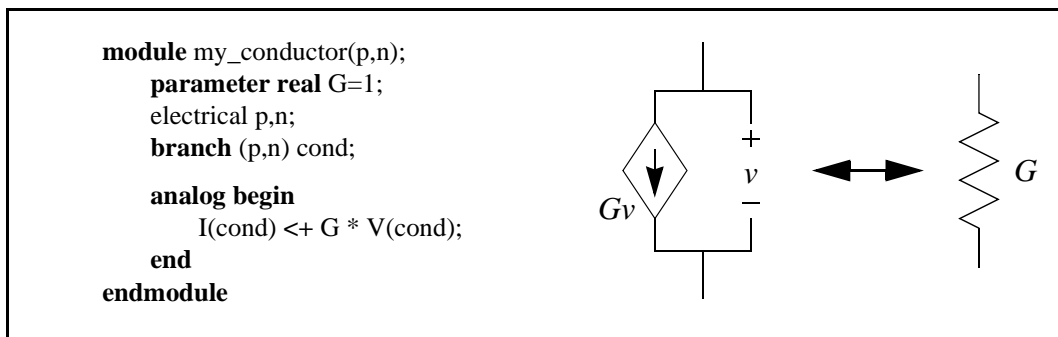


Figure 5-3 Linear conductor model

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

Figure 5-4 shows the model for a linear resistor.

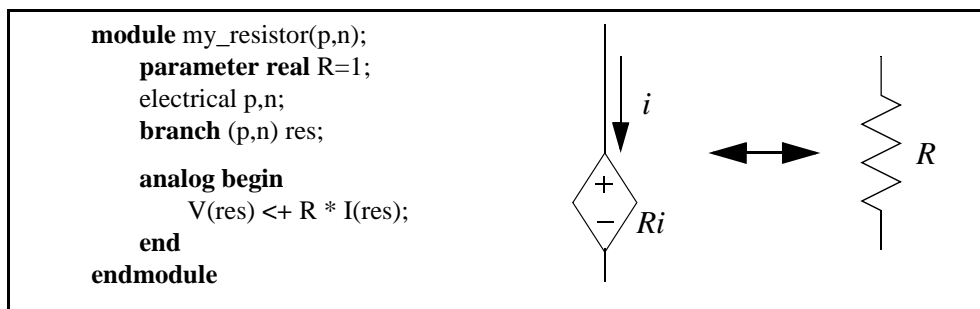


Figure 5-4 Linear resistor model

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 its three components,

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

which can be defined 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 its three components,

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

which can be defined 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-AMS HDL allows components to be described with implicit equations.

Example:

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.

$$I(a, c) <+ is*(\text{limexp}((V(a, c) - rs*I(a, c))/\$vt) - 1);$$

5.1.4 Port branches

The port access function accesses the flow into a port of a module. The name of the access function is derived from the flow nature of the discipline of the port. However ($<>$) is used to delimit the port name, e.g., $I(<a>)$ accesses the current through module port a .

Examples:

Consider rewriting the *junction diode* so 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) i_diode, junc_cap;
  parameter real is = 1e-14, tf = 0, cjo = 0, imax = 1, phi = 0.7 ;

```

```

analog begin
  I(i_diode) <+ is*(limexp(V(i_diode)/$vt) - 1);
  I(junc_cap) <+
    ddt(tf*I(i_diode) - 2*cjo*sqrt(phi*(phi*V(junc_cap))));
  if (I(<a>) > imax)
    $strobe( "Warning: diode is melting!" );
end
endmodule

```

The expression $v(<a>)$ is invalid for ports and nets, where v is a potential access function. The port branch $I(<a>)$ can not be used on the left side of a contribution operator $<+>$.

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 switched branch is shown in Figure 5-5.

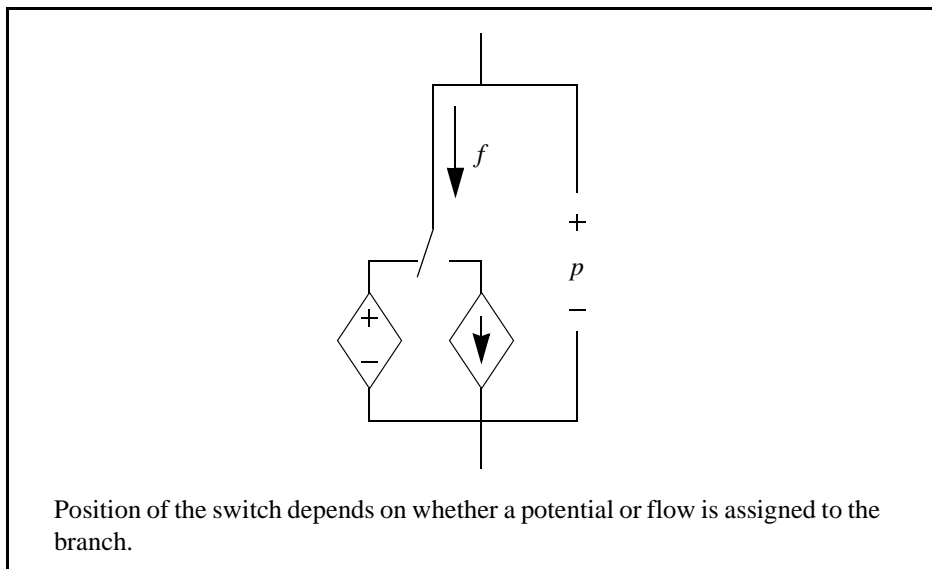


Figure 5-5 Circuit model for a switched source branch

Examples:

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

```

module relay (p, n, ps, ns);
  electrical p, n, ps, ns;
  parameter vth=0.5;
  integer closed;
  analog begin
    closed = (V(ps,ns) >vth ? 1 : 0);

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

```

A discontinuity of order zero (0) 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 (0).

Examples:

```

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

```

is equivalent to

```

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

```

5.2 Signal access for vector branches

Verilog-AMS HDL allows ports, nets, and branches to be arranged as vectors, however, the access functions can only be applied to scalars or individual elements of a vector. The scalar element of a vector is selected with an index, e.g., `V(in[1])` accesses the voltage `in[1]`.

The index must be a genvar expression which is an expression consisting of literals and/or genvar variables. Genvar variables can only be assigned to as the iteration index of for loops; they allow signal access within looping constructs.

Examples:

The following examples illustrate applications of access functions to elements of an analog signal vector or bus. In the `N-bit DAC` example, the analog vector `in` is accessed within an `analog for` loop containing the genvar variable `i`. In the following `fixed-width DAC` example, literal values are used to access elements of the bus directly.


```

//
// N-bit DAC example.
//

module dac(out, in, clk);
parameter integer width = 8 from [2:24];
  parameter real fullscale = 1.0, vth = 2.5, td = 1n, tt = 1n;
  output out;
  input [0:width-1] in;
  input clk;
  electrical out;
  electrical [0:width-1] in;
  electrical clk;

  real aout;
  genvar i;

  analog begin
    @(cross(V(clk) - vth, +1)) begin
      aout = 0;
      for (i = width - 1; i >= 0; i = i - 1) begin
        if (V(in[i]) > vth) begin
          aout = aout + fullscale/pow(2, width - i);
        end
      end
      end
      V(out) <+ transition(aout, td, tt);
    end
  endmodule

//
// 8-bit fixed-width DAC example.
//

module dac8(out, in, clk);
  parameter real fullscale = 1.0, vth = 2.5, td = 1n, tt = 1n;
  output out;
  input [0:7] in;
  input clk;
  electrical out;
  electrical [0:7] in;
  electrical clk;

  real aout;

  analog begin
    @(cross(V(clk) - 2.5, +1)) begin
      aout = 0;
      aout = aout + ((V(in[7]) > vth) ? fullscale/2.0 : 0.0);
      aout = aout + ((V(in[6]) > vth) ? fullscale/4.0 : 0.0);
      aout = aout + ((V(in[5]) > vth) ? fullscale/8.0 : 0.0);
      aout = aout + ((V(in[4]) > vth) ? fullscale/16.0 : 0.0);
    end
  end

```

```

        aout = aout + ((V(in[3]) > vth) ? fullscale/32.0 : 0.0);
        aout = aout + ((V(in[2]) > vth) ? fullscale/64.0 : 0.0);
        aout = aout + ((V(in[1]) > vth) ? fullscale/128.0 : 0.0);
        aout = aout + ((V(in[0]) > vth) ? fullscale/256.0 : 0.0);
    end

    V(out) <+ transition(aout, td, tt);
end

endmodule

```

The syntax for analog signal access is shown in Syntax 5-1.

```

access_function_reference ::=
    bvalue
    | pvalue

bvalue ::=
    access_identifier ( analog_signal_list )

analog_signal_list ::=
    branch_identifier
    | array_branch_identifier [ genvar_expression ]
    | net_or_port_scalar_expression
    | net_or_port_scalar_expression , net_or_port_scalar_expression

net_or_port_scalar_expression ::=
    net_or_port_identifier
    | array_net_or_port_identifier [ genvar_expression ]
    | vector_net_or_port_identifier [ genvar_expression ]

pvalue ::=
    flow_access_identifier (< port_scalar_expression >)

port_scalar_expression ::=
    port_identifier
    | array_port_identifier [ genvar_expression ]
    | vector_port_identifier [ genvar_expression ]

```

Syntax 5-1—Syntax for scalar selection of vector signals

5.2.1 Accessing net and branch signals

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

Examples:

```

electrical out, in ;// as defined in Section 3.4.2.1
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);

```

5.2.2 Accessing attributes

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

Examples:

```

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

analog begin
    I(a,b) <+ c*ddt(V(a,b), a.potential.abstol) ;
    I(cap) <+ c*ddt(V(cap), n1.potential.abstol) ;
end

```

The syntax for referencing access attributes is shown in Syntax 5-2.

```

attribute_reference ::=
    net_identifier . pot_or_flow . attribute_identifier

```

Syntax 5-2—Syntax for referencing attributes of a net

5.3 Contribution statements

Verilog-AMS HDL uses the *branch contribution operator* <+ to describe analog behavior. This operator is only valid within an *analog block*. Branch contribution statements are statements which use the branch contribution operators to describe behavior in terms of a mathematical mapping of input signals to output signals.

5.3.1 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 which evaluates to or can be promoted to a real value. The left-hand side specifies the source branch signal where the right-hand side shall be assigned. It shall consist of a signal access function applied to a branch.

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. The left-hand side can not use a port access function.

Examples:

The following modules model a resistor and a capacitor.

```
module resistor(p, n);
    electrical p, n;
    parameter real r = 0;

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

endmodule

module capacitor(p, n);
    electrical p, n;
    parameter real c = 0;

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

endmodule
```

5.3.1.1 Relations

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

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

Examples:

The following module models a simple single-ended amplifier.

```
module amp(out, in);

    input in;
    output out;
    electrical out, in;
    parameter real Gain = 1;

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

endmodule
```

5.3.1.2 Evaluation

A statement is evaluated as follows for source branch contributions:

1. The simulator evaluates the right-hand side.
2. The simulator adds the value of the right-hand side to any previously retained value of 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.

Parasitics are added to the amplifier shown in Section 5.3.1.1 by simply adding additional contribution statements to model the input admittance and output impedance.

Examples:

```

module amp(out, in);
    inout out, in;
    electrical out, in;
    parameter real Gain = 1, Rin = 1, Cin = 1, Rout = 1, Lout = 1;

    analog begin
        // gain of amplifier
        V(out) <+ Gain*V(in);

        // model input admittance
        I(in) <+ V(in)/Rin;
        I(in) <+ Cin*ddt(V(in));

        // model output impedance
        V(out) <+ Rout*I(out);
        V(out) <+ Lout*ddt(I(out));
    end
endmodule

```

5.3.1.3 Value retention

Contributing a flow to a branch which 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 which already has a value retained for the flow results in the flow being discarded and the branch being converted into a potential source. This is used to model switches. It is illegal to contribute to an external switch branch from within an analog block.

Examples:

```

module switch(p, n, cp, cn);
    electrical p, n, cp, cn;
    parameter real thresh = 0;

```

```

analog begin
    // stop to resolve threshold crossings
    @(cross(V(cp,cn) - thresh, 0));

    if (V(cp,cn) > thresh)
        V(p,n) <+ 0;
    else
        I(p,n) <+ 0;
    end

endmodule

```

The syntax for source contribution statement is shown in Syntax 5-3.

```

analog_branch_contribution ::=
    bvalue <+ analog_expression ;

```

Syntax 5-3—Syntax for branch contribution

5.3.2 Indirect branch assignments

Verilog-AMS HDL allows descriptions which implicitly specify a branch voltage or current in fixed-point form. The branch voltage or current is assigned a value by an expression which uses the branch voltage or current. This occurred in the simple implicit diode model in Section 5.1.3.4, where $I(a, c)$ appeared on both sides of the contribution operator.

Examples:

Consider the model for an ideal *opamp*. In this model, the output is driven to the voltage which results in the input voltage being zero (0). The constitutive equation is

$$V(in) == 0$$

which can be formulated 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 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-AMS HDL includes a special syntax to use in this situation. The above branch contribution can be rewritten using an *indirect branch assignment*:

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

which reads “find $V(out)$ so $V(in) == 0$ ”.

This indicates *out* is driven with a voltage source and the source voltage needs to satisfy the given equation. Any branches referenced in the equation are only probed and not driven. In particular, $V(in)$ acts as a voltage probe.

Examples:

A complete description of an ideal *opamp* is:

```

module opamp(out, pin, nin);
    electrical out, pin, nin;

    analog
        V(out):V(pin,nin) == 0;

endmodule

```

Syntax 5-4 shows the syntax for an indirect assignment statement.

```

analog_indirect_branch_assignment ::=
    bvalue : nexpr == analog_expression ;

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

```

Syntax 5-4—Syntax for indirect branch assignment

5.3.2.1 Multiple indirect assignments

For multiple indirect assignments statements, the targets frequently can be paired with any equation.

Examples:

The following ordinary differential equation,

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

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

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

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.

5.3.2.2 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 `<+.`

It is illegal to indirectly assign to an external branch or contribute to an external branch which has an indirect branch assignment.

Section 6

Analog behavior

The description of an analog behavior consists of setting up contributions (see Section 5.3) for various signals 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 time behavioral definitions within *IEEE 1364-1995 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-AMS HDL allows 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 statements allowed within the analog block are separated into two categories, *analog_statements* and *(non analog) statements*. The *analog_statements* are restricted to the analog block whereas the *(non analog) statements* can appear anywhere within the module scope, including an analog block. The distinction is based upon the visibility and usage of these behavioral constructs within a Verilog-AMS HDL module definition.

The syntax for analog block is shown in Syntax 6-1.

```

analog_block ::=
    analog analog_statement ;

analog_statement ::=
    analog_seq_block
  | analog_branch_contribution
  | analog_indirect_branch_assignment
  | analog_procedural_assignment
  | analog_conditional_statement
  | analog_for_statement
  | analog_case_statement
  | analog_event_controlled_statement
  | system_task_enable
  | statement

statement ::=
  | seq_block
  | procedural_assignment
  | conditional_statement
  | loop_statement
  | case_statement

```

Syntax 6-1—Syntax for analog procedural block

The statements within the analog block are used to define the continuous-time behavior of the module. The behavioral description is a mathematical mapping of input signals to output signals. The mapping is done with contribution statements using the form

```
signal <+ analog_expression ;
```

or by an indirect branch assignment. The *analog_expression* can be any combination of linear, nonlinear, or differential expressions of module signals, constants, and parameters (see Section 5).

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

6.2 Block statements

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

6.2.1 Sequential blocks

The syntax for sequential blocks is shown in Syntax 6-2.

```
seq_block ::=  
    begin [ : block_identifier { block_item_declaration } ]  
        { statement }  
    end  
  
analog_seq_block ::=  
    begin [ : block_identifier { block_item_declaration } ]  
        { analog_statement }  
    end  
  
block_item_declaration ::=  
    parameter_declaration  
    | integer_declaration  
    | real_declaration
```

Syntax 6-2—Syntax for the sequential blocks

An *analog_seq_block* is a *seq_block* which encapsulates one or more *analog_statements*.

6.2.2 Block names

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

6.3 Procedural assignments

In Verilog-AMS HDL, branch contributions and indirect branch assignments are used for modifying signals. Procedural assignments are used for modifying integer and real variables. The syntax for procedural assignments shown in Syntax 6-3.

```

procedural_assignment ::=
    lexpr = expression ;

analog_procedural_assignment ::=
    lexpr = analog_expression ;

lexpr ::=
    integer_identifier
  | real_identifier
  | array_element

array_element ::=
    integer_identifier [ expression ]
  | real_identifier [ expression ]

```

Syntax 6-3—Syntax for procedural assignments

The left-hand side of a procedural assignment shall be an integer or 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.

An *analog_procedural_assignment* is defined as a procedural assignment whose right-hand side *expression* is an *analog_expression* involving analog operators.

6.4 Conditional statement

The *conditional statement* (*if-else* statement) is used to determine whether a statement is executed or not. The syntax of a conditional statement is shown in Syntax 6-4.

```

conditional_statement ::=
    if ( expression ) true_statement_or_null ;
    [ else false_statement_or_null ; ]

```

Syntax 6-4—Syntax of conditional statement

If the expression evaluates to *True* (that is, has a non-zero value), the *true_statement* shall be executed. If it evaluates to *False* (has a zero value (0)), the *true_statement* shall not be executed. If there is an *else false_statement* and expression is *False*, the *false_statement* shall be executed.

Since the numeric value of the *if* expression is tested for being zero (0), certain shortcuts are possible (see Section 4.1).

6.4.1 Examples

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()* which 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* shall be used to force the proper association, as shown below.

```

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

```

Nesting 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 shall be executed and this action shall terminate the whole chain. Each statement is either a single statement or a sequential block of statements.

6.4.2 Analog conditional statements

Analog conditional statements are syntactically equivalent to conditional statements except the *True* and/or *False* statement are *analog_statements*, as shown in Syntax 6-5. The conditional expression shall be a *genvar_expression*. (See the discussion in Section 4.4.1 regarding restrictions on the usage of analog operators.)

```

analog_conditional_statement ::=
    if ( genvar_expression ) true_analog_statement_or_null ;
    [ else false_analog_statement_or_null ; ]

```

Syntax 6-5—Syntax of analog conditional statement

6.5 Case statement

The *case statement* is a multi-way decision statement which tests if an expression matches one of a number of other expressions, and if so, branches accordingly. The case statement has the syntax shown in Syntax 6-6.

```

case_statement ::=
    case ( expression ) case_item { case_item } endcase
    | casex ( expression ) case_item { case_item } endcase
    | casez ( expression ) case_item { case_item } endcase
case_item ::=
    expression { , expression } : statement_or_null
    | default [ : ] statement_or_null

```

Syntax 6-6—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 this 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; otherwise none of the *case_item* statements are executed.

The **casex** and the **casez** versions of the *case* statement are described in Section 8.3.2 and *IEEE 1364-1995 Verilog HDL*.

6.5.1 Analog case statements

Analog case statements are syntactically equivalent to case statements except the case item statements can also be *analog_statements*, as shown in Syntax 6-7. The conditional expression shall be a *genvar* expression. See the discussion in Section 4.4.1 regarding restrictions on the usage of analog operators.

```

analog_case_statement ::=
    case ( genvar_expression ) analog_case_item { analog_case_item } endcase
    | casex ( genvar_expression ) analog_case_item { analog_case_item } endcase
    | casez ( genvar_expression ) analog_case_item { analog_case_item } endcase
analog_case_item ::=
    genvar_expression { , genvar_expression } : analog_statement_or_null
    | default [ : ] analog_statement_or_null

```

Syntax 6-7—Syntax for analog case statement

The **casex** and the **casez** versions of the case statement are described in Section 8.3.2 and *IEEE 1364-1995 Verilog HDL*.

6.5.2 Constant expression in case statement

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

Examples:

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

```
integer encode [2:0];

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
```

The *case* expression here is a *constant* expression (1). The *case_items* are expressions (array elements) and are compared against the constant expression for a match.

6.6 Looping statements

There are several types of looping statements: **repeat()**, **while()**, and **for()**. These statements provide a means of controlling the execution of a statement zero (0), one (1), or more times.

The **for()** looping statements can be used to describe analog behaviors using analog operators.

Analog operators are not allowed in the **repeat()**, **while()**, and **for()** looping statements. They are allowed in *analog_for* and *generate* statements.

6.6.1 Repeat and while statements

repeat() executes a statement a fixed number of times. Evaluation of the 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.

The *repeat* and *while* expressions shall be evaluated once before the execution of any statement in order to determine the number of times, if any, the statements are executed. The syntax for **repeat()** and **while()** statements is shown in Syntax 6-8.


```
repeat_statement ::=
    repeat ( expression ) statement

while_statement ::=
    while ( expression ) statement
```

Syntax 6-8—Syntax for repeat and while statements

6.6.2 For statements

The **for()** statement is a looping construct which controls execution of its associated statement(s) using an index variable. If the associated statement is an *analog_statement*, then the control mechanism shall consist of *genvar_assignments* and *genvar_expressions* to adhere to the restrictions associated with the use of analog operators. If the associated statements are not *analog_statements*, the **for()** statement can use procedural assignments and expressions, including *genvar_expressions*.

The **for()** statement controls execution of its associated statement(s) by a three-step process:

1. it executes an assignment normally used to initialize an integer which controls the number of loops executed.
2. it evaluates an expression—if the result is zero (0), the *for-loop* exits; otherwise, the *for-loop* executes its associated statement(s) and then performs Step 3.
3. it executes an assignment normally used to modify the value of the loop-control variable and repeats Step 2.

Syntax 6-9 shows the syntax for the two forms of the **for()** statements.

```
for_statement ::=
    for ( procedural_assignment ; expression ;
          procedural_assignment ) statement

analog_for_statement ::=
    for ( genvar_assignment ; genvar_expression ;
          genvar_assignment ) analog_statement
```

Syntax 6-9—Syntax for the for statements

analog_for statements are syntactically equivalent to **for()** statements except the associated statement is also an analog statement (which contains analog operations). The analog statement puts the additional restriction upon the procedural assignment and conditional expressions of the *for-loop* to be statically evaluable. Verilog-AMS HDL provides *genvar*-derived expressions for this purpose.

Examples:

```

module genwarexp(out, dt);
  parameter integer width = 1;
  output out;
  input dt[1:width];
  electrical out;
  electrical dt[1:width];
  genvar k;
  real tmp;

  analog begin
    tmp = 0.0;
    for (k = 1; k <= width; k = k + 1) begin
      tmp = tmp + V(dt[k]);
      V(out) <+ ddt(V(dt[k]));
    end
  end
endmodule

```

See the discussion in Section 4.4.1 regarding other restrictions on the usage of analog operators.

6.7 Events

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

- events have no time duration
- events can be triggered and detected in different parts of the behavioral model
- events do not block the execution of an analog block
- events can be detected using the @ operator
- events do not hold any data
- there can be both digital and analog events

There are two types of analog events, *global events* (Section 6.7.4) and *monitored events* (Section 6.7.5). Null arguments are not allowed in analog events.

6.7.1 Event detection

Analog event detection consist of an event expression followed by a procedural statement, as shown in Syntax 6-10.

```

event_control_statement ::=
    event_control statement_or_null

event_control ::=
    @ event_identifier
    | @ ( event_expression )

analog_event_expression ::=
    global_event
    | event_function
    | digital_expression
    | event_identifier
    | posedge digital_expression
    | negedge digital_expression
    | event_expression or event_expression

digital_event_expression ::=
    digital_expression
    | event_identifier
    | posedge digital_expression
    | negedge digital_expression
    | event_function
    | digital_event_expression or digital_event_expression

```

Syntax 6-10—Syntax for event detection

The procedural statement following the event expression is executed whenever the event described by the expression changes. The analog event detection is non-blocking, meaning 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 the **or** operator.

The parenthesis around the event expression are required.

6.7.2 Event OR operator

The “OR-ing” of events indicates the occurrence of any one of the events specified shall trigger the execution of the procedural statement following the event. The keyword **or** is used as an event OR operator.

Examples:

```

analog begin
    @(initial_step or cross(V(smpl)-2.5,+1)) begin
        vout = (V(in) > 2.5);
    end
    V(out) <+ vout;
end

```

Here, **initial_step** is a global event and **cross()** returns a monitored event. The variable `vout` is set to zero (0) or one (1) whenever either event occurs.

6.7.3 Event triggered statements

The following two restrictions apply to statements which are evaluated as a result of an event being triggered.

- The statement can not have expressions which use analog operators. This statement can not maintain its internal state since it is only executed intermittently when the corresponding event is triggered.
- The statement can not be a contribution statement because it can generate discontinuity in analog signals.

6.7.4 Global events

Global events are generated by a simulator at various stages of 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.

Global events are pre-defined in Verilog-AMS HDL. These events can not be redefined in a model.

The pre-defined global events are shown in Syntax 6-11.

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

analysis_list ::=
    analysis_name { , analysis_name }

analysis_name ::=
    " analysis_identifier "
```

Syntax 6-11—Global events

initial_step and **final_step** generate global events on the first and the last point in an analysis respectively. They are useful when performing actions which should only occur at the beginning or the end of an analysis. Both global events can take an optional argument, consisting of an analysis list for the active global event.

Examples:

For example,

```
@(initial_step("ac", "dc"))    // active for dc and ac only
@(initial_step("tran")) // active for transient only
```

Table 6-1 describes the return value of **initial_step** and **final_step** for standard analysis types. Each column shows the return-on-event status. One (1) represents *Yes* and zero (0) represents *No*. A Verilog-AMS HDL simulator can use any or all of these

typical analysis types. Additional analysis names can also be used as necessary for specific implementations. (See Section 4.5.1 for further details.)

Table 6-1—Return Values for initial_step and final_step

Analysis ^a	DCOP OP	TRAN OP p1 pN	AC OP p1 pN	NOISE OP p1 pN
initial_step	1	1 0 0	1 0 0	1 0 0
initial_step("ac")	0	0 0 0	1 0 0	0 0 0
initial_step("noise")	0	0 0 0	0 0 0	1 0 0
initial_step("tran")	0	1 0 0	0 0 0	0 0 0
initial_step("dc")	1	0 0 0	0 0 0	0 0 0
initial_step(<i>unknown</i>)	0	0 0 0	0 0 0	0 0 0
final_step	1	0 0 1	0 0 1	0 0 1
final_step("ac")	0	0 0 0	0 0 1	0 0 0
final_step("noise")	0	0 0 0	0 0 0	0 0 1
final_step("tran")	0	0 0 1	0 0 0	0 0 0
final_step("dc")	1	0 0 0	0 0 0	0 0 0
final_step(<i>unknown</i>)	1	0 0 0	0 0 0	0 0 0

a. pX Table 6-1 designates frequency/time analysis point X, X = 1 to N;
OP designates the Operating Point.

Examples:

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
  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

```

initial_step and **final_step** 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, the simulator generates an event on the first point and the last point of that particular analysis, respectively.

If no analysis list is specified, the **initial_step** global event is active during the solution of the first point (or initial DC analysis) of every analysis. The **final_step** global event, without an analysis list, is only active during the solution of the last point of every analyses.

6.7.5 Monitored events

Monitored events are detected using event functions (see Syntax 6-12) with the @ operator. The triggering of a monitored event is implicit due to change in signals, simulation time, or other runtime conditions.

```

event_function ::=
    cross_function
    | timer_function

```

Syntax 6-12—Monitored events

6.7.5.1 cross function

The **cross()** function is used for generating a monitored analog event to detect threshold crossings in analog signals when the expression crosses zero (0) in the specified direction. In addition, **cross()** controls the timestep to accurately resolve the crossing.

The general form is

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

where *expr* is required, and *dir*, *time_tol*, and *expr_tol* are optional. All arguments are real expressions, except *dir* (which is an integer expression). If the tolerances are not defined, then the tool (e.g., the simulator) sets them. If either or both tolerances are defined, then the direction shall also be defined.

The direction indicator can only evaluate to +1, -1, or 0. If it is set to 0 or is not specified, the event and timestep control occur on both positive and negative crossings of the signal. If *dir* is +1 (or -1), the event and timestep control only occur on rising edge

(falling edge) transitions of the signal. For any other transitions of the signal, the `cross()` function does not generate an event.

expr_tol and *time_tol* are defined as shown in Figure 6-1. They represent the maximum allowable error between the estimated crossing point and the true crossing point.

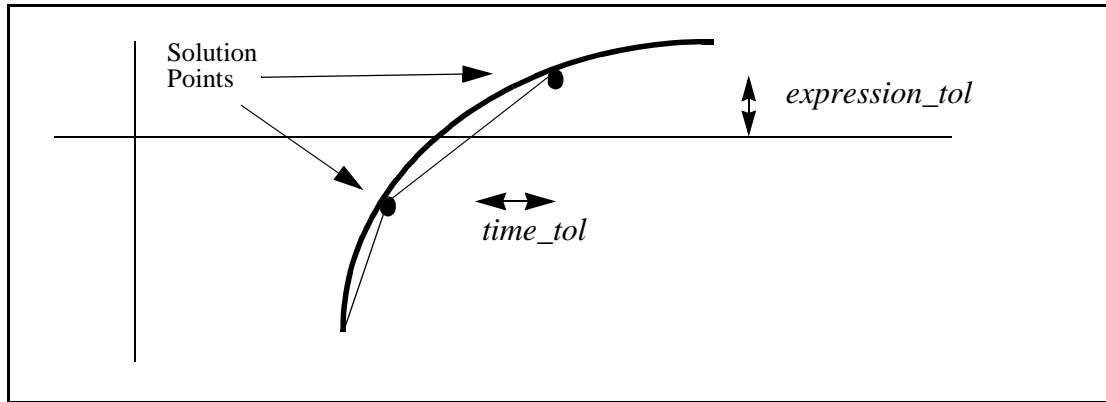


Figure 6-1 Relationship between time tolerance in expression tolerance

If *expr_tol* is defined, *time_tol* shall also be defined and both tolerances shall be satisfied at the crossing.

Examples:

The following description of a sample-and-hold illustrates how the `cross()` function can 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 its internal state and has the same restrictions as analog operators. In particular, it shall not be used inside an `if()`, `case()`, `casex()`, or `casez()` statement unless the conditional expression is a genvar expression. In addition, `cross()` is not allowed in the `repeat()` and `while()` iteration statements. It is allowed in the *analog_for* statements.

6.7.5.2 timer function

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

The general form is

```
timer( start_time [ , period [ , time_tol ] ] );
```

where *start_time* is required; *period* and *time_tol* are optional arguments. All arguments are real expressions.

The **timer()** function schedules an event which occurs at an absolute time (*start_time*). The analog simulator places a time point within *timetol* of an event. At that time point, the event evaluates to *True*.

If *time_tol* is not specified, the default time point is at, or just beyond, the time of the event. If the *period* is specified as greater than zero (0), the timer function schedules subsequent events at multiples of *period*.

Examples:

A pseudo-random bit stream generator is an example how the timer function can 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
```


Section 7

Hierarchical structures

Verilog-AMS 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.

Verilog-AMS HDL provides a mechanism to customize the behavior of embedded modules using parameters. The embedded module parameter default value can be modified through a higher-level module's parameter override or a hierarchy independent `defparam` statement.

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**, as shown in Syntax 7-1. 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 (see Section 7.1.2). The identifiers in this list shall be declared in `input`, `output`, or `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.

The keyword **macromodule** can be used interchangeably with the keyword **module** to define a module. An implementation can choose to treat module definitions beginning with the **macromodule** keyword differently.

```

module_declaration ::=
    module_keyword module_identifier [ digital_list_of_ports ] ;
    [ module_items ]
    endmodule

module_keyword ::=
    module
    | macromodule

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
    | digital_continuous_assignment
    | digital_gate_instantiation
    | digital_udp_instantiation
    | digital_specify_block
    | digital_initial_construct
    | digital_always_construct

```

Syntax 7-1—Syntax for module

The definitions for *module_item_declaration* and *parameter_override* are shown in Syntax 7-2.

The definition of *module_instantiation* is shown in Syntax 7-3.

```

module_item_declaration ::=
    parameter_declaration
  | digital_input_declaration
  | digital_output_declaration
  | digital_inout_declaration
  | ground declaration
  | integer_declaration
  | real_declaration
  | net_discipline_declaration
  | genvar_declaration
  | branch_declaration
  | analog_function_declaration
  | digital_function_declaration
  | digital_net_declaration
  | digital_reg_declaration
  | digital_time_declaration
  | digital_realtime_declaration
  | digital_event_declaration
  | digital_task_declaration

parameter_override ::=
    defparam list_of_param_assignments ;

```

Syntax 7-2—Syntax for module_item_declaration and parameter_override

7.1.1 Top-level modules

Top-level modules are modules which 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.

Syntax 7-3 shows 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 }

named_param_override ::=
    . parameter_identifier ( constant_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 ]

```

Syntax 7-3—Syntax for module instantiation

The following concepts apply:

- 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 element 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 net identifier or a sub-range of a vector net.

Examples:

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;
ground gnd;
electrical cout, inp, inm;
parameter real td = 1n, tr = 1n, tf = 1n;
real vcout;

analog begin
    @(cross(V(inp) - V(inm), 0))
        vcout = ((V(inp) > V(inm)) ? 1 : 0);
    V(vcout) <+ transition(vcout, 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, gnd);
    d2a #(.width(1)) D1(aa2, ref, out); // A D/A converter

endmodule

```

The comparator instance C1 and the integrator instance I1 in Figure 7-1 use named port connections, whereas the comparator instance C2 and the d2a (not described here) instance D1 use ordered port connections. Note the integrator instance I1 overrides gain parameter positionally, whereas the d2a instance D1 overrides width parameter by named association.

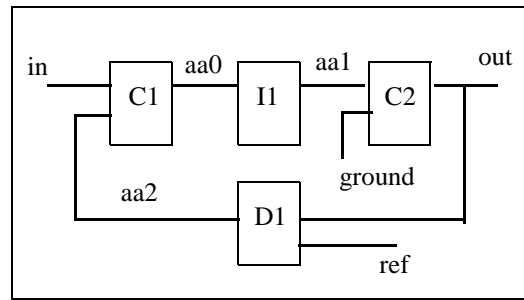


Figure 7-1 Comparator and integrator modules

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 two ways to alter parameter values: the *defparam* statement, which allows assignment to parameters using their hierarchical names, and the *module instance parameter value assignment*, which allows values to be assigned inline during module instantiation. If a *defparam* assignment conflicts with a module instance parameter, the parameter in the module shall take the value specified by the *defparam*.

The module instance parameter value assignment comes in two forms, by ordered list or by name. The first form is *module instance parameter value assignment by order*, which allows values to be assigned in-line during module instantiation in the order of their declaration. The second form is *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 details about hierarchical names.

The expression on the right-hand side of a *defparam* assignments shall be a constant expression involving only constant numbers and references to parameters. The referenced parameters (on the right-hand side of a *defparam*) shall 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. Its syntax is shown in Syntax 7-4.

```

parameter_override ::=
    defparam list_of_defparam_assignments ;

list_of_defparam_assignments ::=
    defparam_assign
    | list_of_defparam_assignments , defparam_assign

defparam_assign ::=
    parameter_identifier = constant_expression
    | parameter_array_identifier [ range ] = constant_param_arrayinit

constant_param_arrayinit ::=
    { param_arrayinit_element_list }

param_arrayinit_element_list ::=
    param_arrayinit_element
    | param_arrayinit_element_list , param_arrayinit_element

param_arrayinit_element ::=
    constant_expression
    | { replicator_constant_expression {constant_expression} }

```

Syntax 7-4—Syntax for *defparam**Examples:*

```

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);
    inout source, drain, gate;
    electrical 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);
    inout source, drain, gate;
    electrical 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

The order of the assignments in module instance parameter value assignment shall 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, the left-most parameter assignment(s) can not be skipped. Therefore, to assign values to a subset of the parameters declared within a module, the declarations of the parameters which make up this subset shall 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 which do not need new values.

Examples:

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

```

module m;
    electrical 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

Parameter assignment by name consists of explicitly linking the parameter name and its value. The name of the parameter shall be the name specified in the instantiated module. It is not necessary to assign values to all the parameters within a module when using this method. Only those parameters which are assigned new values need to be specified.

The parameter expression is optional so the instantiating module can document the existence of a parameter without assigning anything to it. The parentheses are required and in this case the parameter retains its default value. Once a parameter is assigned a value, there shall not be another assignment to this parameter name.

Example:

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

```

module n(lo_out, rf_in):
  electrical lo_out, rf_in;

  //create an instance and set parameters
  vco #(.centerFreq(5000), .convGain(1000)) vco1(lo_out, rf_in);
endmodule

```

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 port and `rf_in` as the second port of `vco1`.

7.2.4 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`.

Examples:

In the following parameter declaration, an update of `gate_width`, whether by a *defparam* statement or in an instantiation statement for the module which 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 nets of module A.

7.3.1 Port association

The syntax for a port association is shown in Syntax 7-5.

```

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

```

Syntax 7-5—Syntax for port

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

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

The two types of module port definitions cannot be mixed; the ports of a particular module definition shall all be defined by order or by name. The port expression is optional because ports can be defined which 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, as shown in Syntax 7-6. 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.)

```

net_discipline_declaration ::=
    discipline_identifier [ range ] list_net_identifiers;

list_net_identifiers ::=
    net_identifier { , net_identifier }

```

Syntax 7-6—Syntax for port type declarations

7.3.2.2 Port direction

Each port listed in the list of ports for the module definition shall be declared in the body of the module as an **input**, **output**, or **inout** (*bidirectional*). This is in addition to any other declaration for a particular port — for example, a *net_discipline*, *reg*, or *wire*. The syntax for port declarations is shown in Syntax 7-7.

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

output_declaration ::=
    output [ range ] list_of_port_identifiers ;

inout_declaration ::=
    inout [ range ] list_of_port_identifiers ;
```

Syntax 7-7—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 shall be identical.

Note: Implementations can limit the maximum number of ports in a module definition, but this shall be a minimum of 256 ports per implementation.

7.3.3 Real valued ports

Verilog-AMS HDL supports ports which are declared to be real-valued and have a discrete-time discipline. This is done using the net type *wreal* (defined in Section 3.5). There can be a maximum of one driver of a real-valued net.

Examples:

```
module top();
    wreal stim;
    reg clk;
    wire [1:8] out;

    testbench tb1 (stim, clk);
    a2d dut (out, stim, clk);

    initial clk = 0;
    always #1 clk = ~clk;
endmodule

module testbench(wout, clk);
    output wout;
    input clk;
    real out;
    wire clk;
    wreal wout;
    assign wout = out;
```

```

    always @(posedge clk) begin
        out = out + $abstime;
    end
endmodule

module a2d(dout, in, clk);
    output [1:8] dout;
    input in, clk;
    wreal in;
    wire clk;
    reg [1:8] dout;
    real residue;
    integer i;

    always @(negedge clk) begin
        residue = in;
        for (i = 8; i >= 1; i = i - 1) begin
            if (residue > 0.5) begin
                dout[i] = 1'b1;
                residue = residue - 0.5;
            end else begin
                dout[i] = 1'b0;
            end
            residue = residue*2;
        end
    end
endmodule

```

7.3.4 Connecting module ports by ordered list

One way to connect the ports listed in a module instantiation with the ports defined by the instantiated module is via an ordered list—that is, the ports listed for the module instance shall be in the same order as the ports listed in the module definition.

Examples:

```

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 lol (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

```

7.3.5 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 — specify 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 following rules apply:

- The name of port shall be the name specified in the module definition.
- The name of port cannot be a bit select or a part select.
- The port expression shall be the name used by the instantiating module and can be one of the following:
 - a simple net identifier
 - a scalar member of a vector net or port declared within the module
 - a sub-range of a vector net or port declared within the module
 - a vector net formed as a result of the concatenation operator
- The port expression is optional so 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 shall be all by order or all by name.

Examples:

```

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 hil (out[1], r, in) ; // adc is same as defined in Section 7.3.4
adc lol (out[0], remainder, r) ;
endmodule

```

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

7.3.6 Port connection rules

All digital ports connected to a net shall be of compatible disciplines, as shall all analog ports connected to a net. Ports of both analog and digital discipline may be connected to a net provided the appropriate connect statements exist (see Section 8.7).

7.3.6.1 Matching size rule

A scalar port can be connected to a scalar net and a vector port can be connected to a vector net or concatenated net expression of the matching width. In other words, the sizes of the ports and net need to match.

7.3.6.2 Resolving discipline of undeclared interconnect signal

Verilog-AMS HDL supports undeclared interconnects between module instances when describing hierarchical structures. That is, a signal appearing in the connection list of a module instantiation need not appear in any port declaration or discipline declaration (see Section 8.4).

7.3.7 Inheriting port natures

A net of continuous discipline shall have a potential nature and may have a flow nature. Because of hierarchical connections, an analog node may be associated with a number of analog nets, and thus, a number of continuous disciplines. The node shall be treated

as having a potential **abstol** with a value equal to the smallest **abstol** of all the potential natures of all the disciplines with which it is associated. The node shall be treated as having a flow **abstol** with a value equal to the smallest **abstol** of all the flow natures, if any, of all the disciplines with which it is associated.

7.4 Hierarchical names

Every identifier in Verilog-AMS HDL 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 as a scope (of a particular branch of the tree).

At the top of the name hierarchy are the names of modules where no instances have been created. This 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 net in the hierarchical name tree is treated as a separate scope with respect to identifiers. A particular identifier can be declared only 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 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 which starts at the level where the path is being used.

Examples:

```

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 7-2 illustrates the hierarchy implicit in the example code.

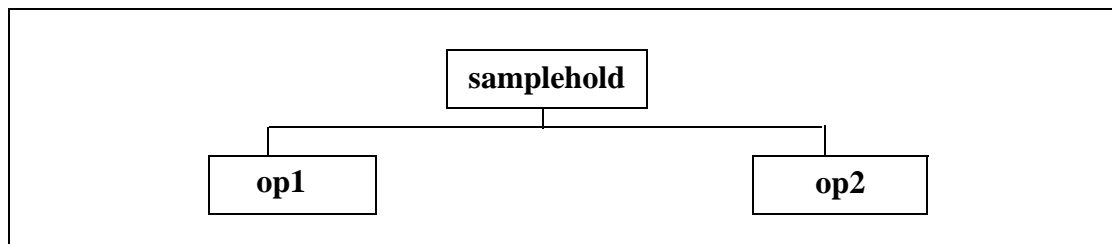


Figure 7-2 Hierarchy in a model

Figure 7-3 is a list of the hierarchical forms of the names of all the object defined in the example code.

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 7-3 Hierarchical path names in a design

From within an analog block, it is possible to use hierarchical name referencing to access signals on an external branch, but not external analog variables or parameters. When accessing external branches, a branch signal (its potential or flow) can be monitored (probed); for source branches, contributions can be made to the output signal.

7.5 Scope rules

The following elements define a new scope in Verilog-AMS HDL:

- modules
- tasks
- named blocks

- functions
- analog functions

An identifier can be used to declare only one item within a scope. This means it is illegal to declare two or more variables which have the same name or to give an instance the same name as the name of the net connected to its output.

If an identifier is referenced directly (without a hierarchical path) within a named block, it shall be declared either locally within the named block, within a module, or within a named block which is higher in the same branch of the name tree containing the named block. If it is declared locally, the local item shall be used; if not, the search shall 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.

Note: Because of the upward searching process, path names which are not strictly on a downward path can be used.

Section 8

Mixed signal

8.1 Introduction

With the mixed use of digital and analog simulators, a common terminology is needed. This section provides the core terminology used in this LRM and highlights the behavior of the mixed-signal capabilities of Verilog-AMS HDL.

Verilog-AMS HDL provides the ability to accurately model analog, digital, and mixed-signal blocks. Mixed-signal blocks provide the ability to access data and be controlled by events from the other domain. In addition to providing mixed-signal interaction directly through behavioral descriptions, Verilog-AMS HDL also provides a mechanism for the mixed-signal interaction between modules.

Verilog-AMS HDL is a hierarchical language which enables top-down design of mixed-signal systems. Connect modules are used in the language to resolve the mixed-signal interaction between modules. These modules can be manually inserted (by the user) or automatically inserted (by the simulator) based on rules provided by the user.

Connect rules and the discipline of the mixed signals can be used to control auto-insertion throughout the hierarchy. Prior to insertion, all net segments of a mixed signal shall first be assigned a discipline. This is commonly needed for interconnect, which often does not have a discipline declared for it. Once a discipline has been assigned (usually through use of a discipline resolution algorithm), connect modules shall be inserted based on the specified connect rules. *Connect rules* control which connect modules are used and where are they inserted.

Connect modules are a special form of a mixed-signal module which provide significant power in accurately modeling the interfaces between analog and digital blocks. They help ensure the drivers and receivers of a connect module are correctly handled so the simulation results are not impacted.

This section also details a feature which allows analog to accurately model the effects the digital receivers for mixed signals containing both drivers and receivers. In addition, special functions provide access to driver values so a more accurate connect module can be created.

The following subsections define these capabilities in more detail.

8.2 Fundamentals

The most important feature of Verilog-AMS HDL is it puts capabilities of both analog and digital modeling into a single language. This section describes how the continuous (analog) and discrete (digital) domains interact together, as well as the mixed-signal-specific features of the language.

8.2.1 Domains

The domain of a value refers to characteristics of the computational method used to calculate it. In Verilog-AMS HDL, a variable is calculated either in the *continuous* (analog) domain or the *discrete* (digital) domain every time. The potentials and flows described in natures are calculated in the continuous domain, while register contents and the states of gate primitives are calculated in the discrete domain. The values of real and integer variables can be calculated in either the continuous or discrete domain depending on how their values are assigned.

Values calculated in the discrete domain change value instantaneously and only at integer multiples of a minimum resolvable time. For this reason, the derivative with respect to time of a digital value is always zero (0). Values calculated in the continuous domain, on the other hand, are continuously varying.

8.2.2 Contexts

Statements in a Verilog-AMS HDL module description can appear in the body of an analog block, in the body of an **initial** or **always** block, or outside of any block (in the body of the module itself). Those statements which appear in the body of an analog block are said to be in the *continuous* (analog) context; all others are said to be in the *discrete* (digital) context. A given variable can be assigned values only in one context or the other, but not in both. The domain of a variable is that of the context from which its value is assigned.

8.2.3 Nets, nodes, ports, and signals

In Verilog-AMS HDL, hierarchical structures are created when higher-level modules create instances of lower level modules and communicate with them through input, output, and bidirectional ports. A *port* represents the physical connection between an expression in the instantiating or parent module and an expression in the instantiated or child module. The expressions involved are referred to as *nets*, although they can include registers, variables, and nets of both continuous and discrete disciplines. A port of an instantiated module has two nets, the upper connection (`vpiHiConn`) which is a net in the instantiating module and the lower connection (`vpiLoConn`) which is a net in the instantiated module, as shown in Figure 8-1. The `vpiLoConn` and `vpiHiConn`

connections to a port are frequently referred to as the *formal* and *actual connections* respectively.

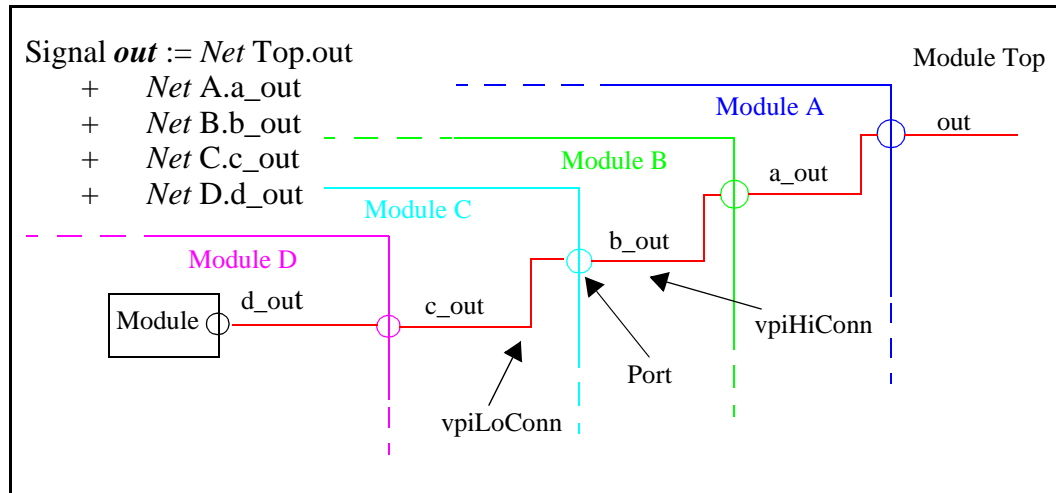


Figure 8-1 Signal “out” hierarchy of net segments

A net can be declared with either a discrete or analog *discipline* or no *discipline* (neutral interconnect). Within the Verilog-AMS language, only digital blocks and primitives can drive a discrete net (*drivers*), and only analog blocks can contribute to an analog net (*contributions*). A *signal* is a hierarchical collection of nets which, because of port connections, are contiguous. If all the nets that make up a signal are in the discrete domain, the signal is a *digital signal*. If all the nets that make up a signal are in the continuous domain, the signal is an *analog signal*. A signal that consists of nets from both domains is called a *mixed signal*.

Similarly, a port whose connections are both analog is an *analog port*, a port whose connections are both digital is a *digital port*, and a port whose connections are analog and digital is a *mixed port*.

Since it is physically one wire in the design, Kirchoff’s current law applies to the whole signal, and it forms one node in analog simulation (see Section 3.4). Drivers in the digital domain are converted to contributions in analog domain using auto-inserted digital-to-analog connection modules (D2As), and the signal value is calculated in the analog domain. Instead of determining the final digital receiver value of the signal by resolving all the digital drivers, the resolved analog signal is converted back to a digital value. A digital behavioral block that reads the value of a signal is a *receiver*, but since Verilog-AMS has no syntax that identifies multiple receivers within a module as distinct, the associated net can be viewed as a single receiver for the purposes of analog to digital conversion. Drivers are created by declaring a reg, instantiating a digital primitive or using a continuous assign statement. Since it is only possible to insert connect modules at port boundaries, when multiple continuous assign statements exist in a module, they are handled by a single connect module.

The drivers and receivers of a mixed signal are associated with their locally-declared net; the discipline of that net is used to determine which connection modules to use. The discipline of the whole signal is found by discipline resolution, as described in Section 8.4, and is used to determine the attributes of the node in simulation.

8.2.4 Mixed-signal and net disciplines

One job of the discipline of a continuous net is to specify the tolerance (*abstol*) for the potential of the associated node. A mixed signal can have a number of compatible continuous nets, with different continuous disciplines and different *abstols*. In this case, the *abstol* of the associated node shall be the smallest of the *abstols* specified in the disciplines associated with all the continuous nets of the signal.

If an undeclared net segment has multiple compatible disciplines connected to it, a connect statement shall specify which discipline to use during discipline resolution.

8.3 Behavioral interaction

Verilog-AMS HDL supports several types of block statements for describing behavior, such as `analog` blocks, `initial` blocks, and `always` blocks. Typically, non-analog behavior is described in `initial` and `always` blocks, assignment statements, or assign declarations. There can be any number of `initial` and `always` blocks in a particular Verilog-AMS HDL module. However there can only be one `analog` block in that module.

Nets and variables in the continuous domain are termed *continuous nets* and *continuous variables* respectively. Likewise nets, regs and variables in the discrete domain are termed *discrete nets*, *discrete regs*, and *discrete variables*. In Verilog-AMS HDL, the nets and variables of one domain can be referenced in the other's context. This is the means for passing information between two different domains (continuous and discrete). Read operations of nets and variables in both domains are allowed from both contexts. Write operations of nets and variables are only allowed from the context of their domain.

Verilog-AMS HDL provides ways to:

- access discrete primaries (e.g., nets, regs, or variables) from a continuous context
- access continuous primaries (e.g., flows, potentials, or variables) from a discrete context
- detect discrete events in a continuous context
- detect continuous events in a discrete context

The specific time when an event from one domain is detected in the other domain is subject to the synchronization algorithm described in Section 8.3.6 and Section 9. This

algorithm also determines when changes in nets and variables of one domain are accessible in the other domain.

8.3.1 Accessing discrete nets and variables from a continuous context

Discrete nets and variables can be accessed from a continuous context. However, because the data types which are supported in continuous contexts are more restricted than those supported in discrete contexts, certain discrete types can not be accessed in a continuous context.

Table 8-1 lists how the various discrete net/variable types can be accessed from a continuous context.

Table 8-1—Discrete net/reg/variable access from a continuous context

Discrete net/reg/ variable type	Examples	Equivalent continuous variable type	Access to this discrete net/reg/variable type from a continuous context
real	real r; real rm[0:8];	real	Discrete reals are accessed in the continuous context as real numbers.
integer	integer i; integer im[0:4];	integer	Discrete integers are accessed in continuous context as integer numbers.
bit	reg r1; wire w1; reg [0:9] r[0:7]; reg r[0:66]; reg [0:34] rb;	integer	Discrete bit and bit groupings (buses and part selects) are accessed in the continuous context as integer numbers. The sign bit (bit 31) of the integer is always set to zero (0). The lowest bit of the bit grouping is mapped to the 0th bit of the integer. The next bit of the bus is mapped to the 1st bit of the integer and so on. If the bus width is less than 31 bits, the higher bits of the integer are set to zero (0). Access of discrete bit groupings with greater than 31 bits is illegal.

The syntax for a Verilog-AMS HDL primary is defined in Syntax 8-1.


```

primary ::=
    number
  | identifier
  | identifier [ expression ]
  | identifier [ msb_constant_expression : lsb_constant_expression ]
  | concatenation
  | analog_function_call
  | string
  | access_function

```

Syntax 8-1—Syntax for primary

Examples:

The following example accesses the discrete primary `in` from a continuous context.

```

module onebit_dac (in, out);
input in;
inout out;
wire in;
electrical out;
real x;

analog begin
    if (in == 0)
        x = 0.0;
    else
        x = 3.0;
        V(out) <+ x;
end

endmodule

```

8.3.2 Accessing X and Z bits of a discrete net in a continuous context

Discrete nets can contain bits which are set to x (*unknown*) or z (*high impedance*). Verilog-AMS HDL supports accessing of 4-state logic values within the analog context. The x and z states must be translated to equivalent analog real or integer values before being used within the analog context. The language supports the following specific features, which provide a mechanism to perform this conversion.

- the case equality operator (`===`)
- the case inequality operator (`!==`)
- the `case`, `casex`, and `casez` statements

- binary, octal and hexadecimal numeric constants which can contain x and z as digits.

The case equality and case inequality operators have the same precedence as the equality operator.

Example:

```

module a2d(dnet, anet);
input dnet;
wire dnet;
logic dnet;
output anet;
electrical out;

analog begin
  case (dnet)
    1'b1:var = 5;
    1'bx:var = var;// hold value
    1'b0:var = 0;
    1'bz:var = 2.5; // high impedance - float value
  endcase
  V(anet) <+ var;

end
endmodule

```

Note: A case statement may be replaced with an *if-else-if* statement using the case equality operators to perform the 4-state logic value comparisons.

Accessing digital net and digital binary constant operands are supported within analog context expressions. It is an error if these operands return 'x' or 'z' bit values when solved. It will be an error if the value of the digital variable being accessed in the analog context goes either to 'x' or 'z'.

Example:

```

module converter(dnet,anet);
reg dnet;
electrical anet;
integer var1;
real var2;

initial begin
  dnet = 1'b1;
  #50 dnet = 1'bz;
  $finish;
end

analog begin
  var1 = 1'bx;// error
  var2 = 1'bz;// error
  var1 = 1 + dnet;// error after #50

```

```

    if (dnet == 1'bx) // error
        $display("Error to access x bit in continuous context");
    V(anet) <+ 1'bz; // error
    V(anet) <+ 1'bz; // error after #50
end
endmodule

```

The syntax for the features that support x and z comparisons in a continuous context is defined in Section 2.5 and Section 6.5. Support for x and z is limited in the analog blocks as defined above.

Note: Consult *IEEE 1364-1995 Verilog HDL* for a description of the semantics of these operators.

8.3.2.1 Special floating point values

Floating point arithmetic can produce special values representing plus and minus infinity and Not-a-Number (NaN) to represent a bad value. While use of these special numbers in digital expressions is not an error, it is illegal to assign these values to a branch through contribution in the analog context.

8.3.3 Accessing continuous nets and variables from a discrete context

All continuous nets can be probed from a discrete context using access functions. All probes which are legal in a continuous context of a module are also legal in the discrete context of a module. Therefore for Verilog-AMS HDL, the definition of *IEEE 1364-1995 Verilog HDL's primary* is shown in Syntax 8-2.

```

digital_primary ::=
    digital_number
    | identifier
    | identifier [ digital_expression ]
    | identifier [ digital_msb_constant_expression : digital_lsb_constant_expression ]
    | digital_concatenation
    | digital_multiple_concatenation
    | digital_function_call
    | ( digital_mintypmax_expression )
    | access_function_reference

```

Syntax 8-2—Syntax for *digital_primary*

Examples:

The following example accesses the continuous net `v(in)` from the discrete context is.

```

module sampler (in, clk, out);
inout in;
input clk;
output out;
electrical in;
wire clk;
reg out;

always @(posedge clk)
    out = V(in);

endmodule

```

Continuous variables can be accessed for reading from any discrete context in the same module where these variables are declared. Because the discrete domain can fully represent all continuous types, a continuous variable is fully visible when it is read in a discrete context.

8.3.4 Detecting discrete events in a continuous context

Discrete events can be detected in a Verilog-AMS HDL continuous context. The arguments to discrete events in continuous contexts are in the discrete context. A discrete event in a continuous context is non-blocking like the other event types allowed in continuous contexts. The syntax for events in a continuous context is shown in Syntax 8-3.

```

event_control_statement ::=
    event_control statement_or_null

event_control ::=
    @ event_identifier
    | @ ( event_expression )

event_expression ::=
    global_event
    | event_function
    | digital_expression
    | event_identifier
    | posedge digital_expression
    | negedge digital_expression
    | event_expression or event_expression

```

Syntax 8-3—Syntax for event control statement

Examples:

The following example shows a discrete event being detected in an analog block.

```

module sampler3 (in, clk1, clk2, out);
input in, clk1, clk2;
output out;
wire clk1;
electrical in, clk2, out;

analog begin
    @(posedge clk1 or cross(V(clk2), 1))
        vout = V(in);
    V(out) <+ vout;
end

endmodule

```

8.3.5 Detecting continuous events in a discrete context

In Verilog-AMS HDL, monitored continuous events can be detected in a discrete context. The arguments to these events are in the continuous context. A continuous event in a discrete context is blocking like other discrete events. For Verilog-AMS HDL, the definition of *IEEE 1364-1995 Verilog HDL's event_expression* is shown in Syntax 8-4.

```

digital_event_expression ::=
    digital_expression
  | event_identifier
  | posedge digital_expression
  | negedge digital_expression
  | event_function
  | digital_event_expression or digital_event_expression

```

Syntax 8-4—Syntax for digital event expression

Examples:

The following example detects a continuous event in an always block.

```

module sampler2 (in, clk, out);
input in, clk;
output out;
wire in;
reg out;
electrical clk;

always @(cross(V(clk) - 2.5, 1))
    out = in;

endmodule

```

8.3.6 Concurrency

Verilog-AMS HDL provides synchronization between the continuous and discrete domains. Simulation in the discrete domain proceeds in integer multiples of the digital tick. This is the smallest value of the second argument of the ``timescale` directive (see Section 16.7 in *IEEE 1364-1995 Verilog HDL*). Thus, values calculated in the digital domain shall be constant between digital ticks and can only change at digital ticks.

Simulation in the continuous domain appears to proceed continuously. Thus, there is no time granularity below which continuous values can be guaranteed to be constant.

The rest of this section describes synchronization semantics for each of the four types of mixed-signal behavioral interaction. Any synchronization method can be employed, provided the semantics preserved. A typical synchronization algorithm is described in Section 9.2.

8.3.6.1 Analog event appearing in a digital event control

In this case, an analog event, such as `cross` or `timer`, appears in an `@()` statement in the digital context.

Examples:

```
always begin
    @(cross(V(x) - 5.5, 1))
    n = 1;
end
```

When it is determined the event has occurred in the analog domain, the statements under the event control shall be scheduled in the digital domain at the largest digital time tick smaller than or equal to the time of the analog event. This event shall not be scheduled in the digital domain earlier than the current digital event (see Section 9.2.3).

8.3.6.2 Digital event appearing in an analog event control

Examples:

```
analog begin
    @(posedge n)
    r = 3.14
end
```

In this case, a digital event, such as `posedge` or `negedge`, appears in an `@()` statement in the analog context.

When it is determined the event has occurred in the digital domain, the statements under the event control shall be executed in the analog domain at the time corresponding to a real promotion of the digital time (e.g., 27 ns to 27.0e-9).

8.3.6.3 Analog primary appearing in a digital expression

In this case, an analog primary (variable, potential, or flow) whose value is calculated in the continuous domain appears in an expression which is in the digital context; thus the analog primary is evaluated in the digital domain.

The expression shall be evaluated using the analog value calculated for the time corresponding to a real promotion of the digital time at which the expression is evaluated.

8.3.6.4 Digital primary appearing in an analog expression

In this case, a digital primary (reg, wire, integer, etc.) whose value is calculated in the discrete domain appears in an expression which is in the analog context; thus the analog primary is evaluated in the continuous domain.

The expression shall be evaluated using the digital value calculated for the greatest digital time tick which is less than or equal to the analog time when the expression is evaluated.

8.3.7 Function calls

Analog functions can only be called from a continuous context. Digital functions can only be called from a digital context.

8.4 Discipline resolution

In general a mixed signal is a collection of nets, some with discrete discipline(s) and some with continuous discipline(s). Additionally, some of the nets can have undeclared discipline(s). Discipline resolution assigns disciplines to those nets whose discipline is undeclared. This is done to control auto-insertion of connect modules, according to the rules embodied in *connect statements*.

The assignments are based on: discipline declarations, ``default_discipline` directives (see Section 3.6), and the hierarchical connectivity of the design. Once all net segments of every mixed signal has been resolved, insertion of connect modules shall be performed.

8.4.1 Compatible discipline resolution

One factor which influences the resolved discipline of a net whose discipline is undeclared is the disciplines of nets to which it is connected via ports; i.e., if multiple compatible disciplines are connected to the same net via multiple ports only one discipline can be assigned to that net. This is controlled by the `resolveto` form of the connect statement; the syntax of this form is described in Section 8.7.2.

If disciplines at the lower connections of ports (where the undeclared net is an upper connection) are among the disciplines in `discipline_list`, the `result_discipline` is the discipline

which is assigned to the undeclared net. If all the nets are of the same discipline, no rule is needed; that discipline becomes the resolved discipline of the net.

Example:

In the example shown in Figure 8-2, *NetA* and *NetB* are undeclared interconnects. *NetB* has *cmos3* and *cmos4* at the lower connection ports, while it is an upper connection.

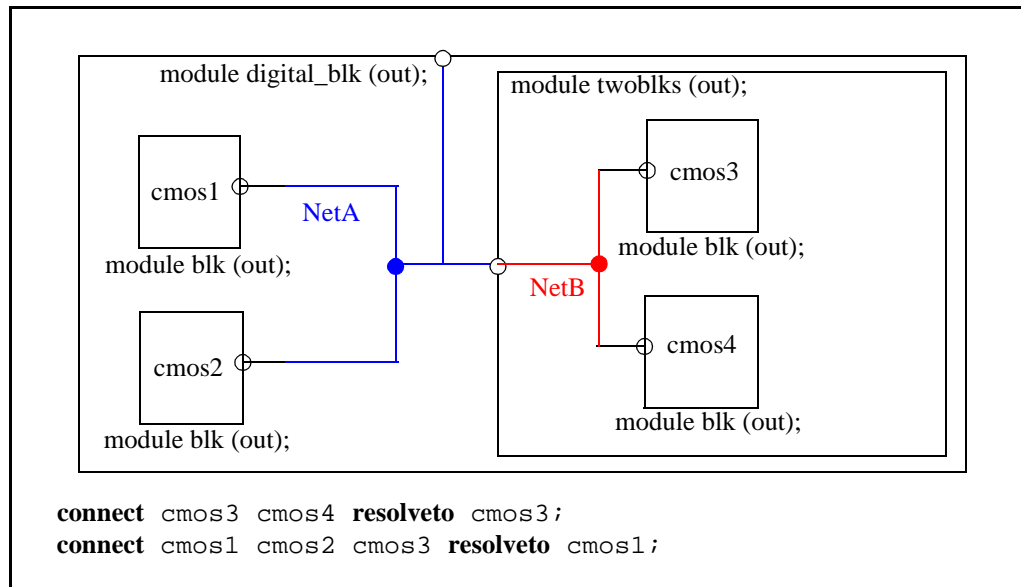


Figure 8-2 Compatible discipline resolution

The first connect statement resolves *NetB* to be assigned the discipline *cmos3*.

NetA has *cmos1*, *cmos2* and the resulting *cmos3* from module *twoblks* at the lower connection ports; based on the second connect statement, it resolves to be assigned the discipline *cmos1*.

8.4.2 Connection of discrete-time disciplines

Ports of discrete-time disciplines (ports where digital signals appear at both upper (*vpiHiConn*) and lower (*vpiLoConn*) connections) shall obey the rules imposed by *IEEE 1364-1995 Verilog HDL* on such connections.

In addition, the real-value nets shall obey the rules imposed by Section 3.5.

8.4.3 Connection of continuous-time disciplines

Ports of continuous-time disciplines (ports where analog signals appear at both upper (*vpiHiConn*) and lower (*vpiLoConn*) connections) shall obey the rules imposed in Section 3.8. It shall be an error to connect incompatible continuous disciplines together.

8.4.4 Resolution of mixed signals

Once discipline declarations and the ``default_discipline` compiler directive have been applied, if any mixed-signal nets are still undeclared additional resolution is needed. This section provides an additional method for discipline resolution of remaining undeclared nets (to control the auto-insertion of connect modules).

There are two modes for this method of resolution, *basic* (the default) and *detail*, which determine how known disciplines are used to resolve these undeclared nets. For the entire design, undeclared nets shall be resolved at each level of the hierarchy where continuous (analog) has precedence over discrete (digital). The selection of these discipline resolution modes shall be vendor-specific.

More than one conflicting discipline declaration from the same context (in or out of context) for the same hierarchical segment of a signal is an error. In this case, *conflicting* simply means an attempt to declare more than one discipline regardless of whether the disciplines are compatible or not.

Sample algorithms for the complete discipline resolution process are listed in Annex F.

8.4.4.1 Basic discipline resolution algorithm

In this mode (the default), both continuous and discrete disciplines propagate up the hierarchy to meet one another. At each level of the hierarchy where continuous and discrete meet for an undeclared net that net segment is declared continuous. This typically results in connect modules being inserted higher up the design hierarchy.

Examples:

In the example shown in Figure 8-3, `NetA`, `NetB`, `NetC`, and `NetD` are undeclared interconnects.

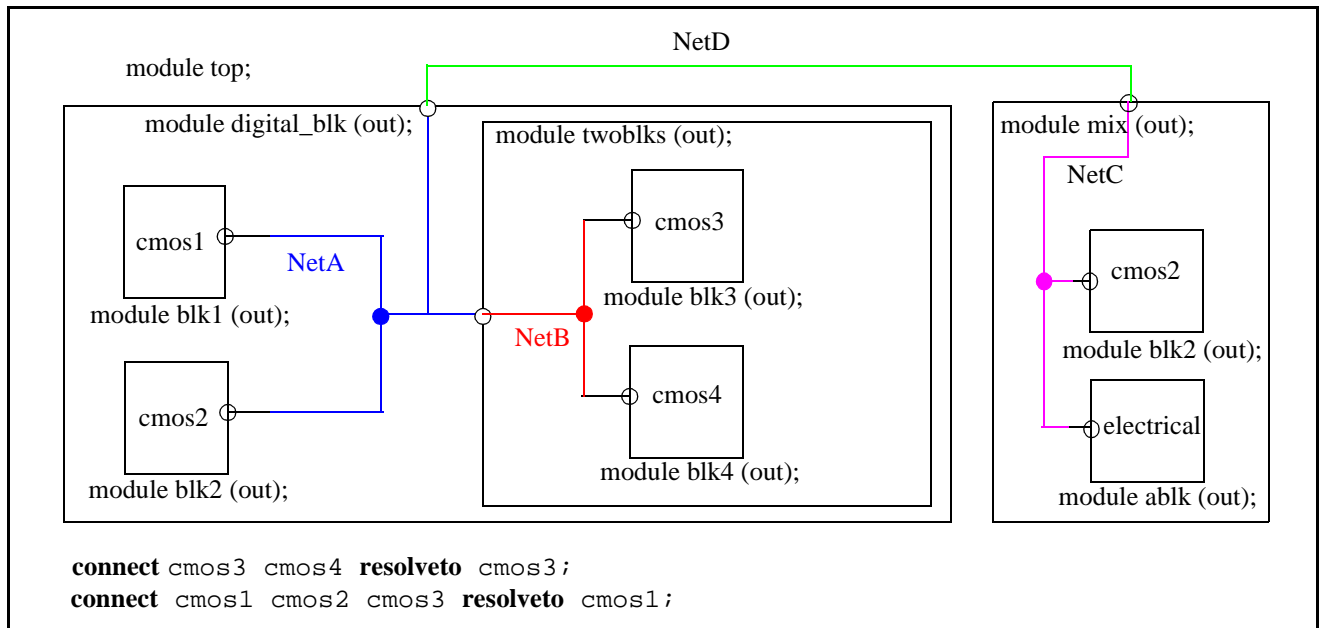


Figure 8-3 Discipline resolution mode: basic

Using the basic mode of discipline resolution and the specified **resolveto** connect statements for this example results in the following:

- NetB resolves to `cmos3` based on the first **resolveto** connect statement.
- NetA resolves to `cmos1` based on the second **resolveto** connect statement.
- NetC resolves to `electrical` based on continuous (`electrical`) winning over discrete (`cmos2`).
- NetD resolves to `electrical` based on continuous (`electrical`) winning over discrete (`cmos1`).

8.4.4.2 Detail discipline resolution algorithm

In this mode continuous disciplines propagate up and then back down to meet discrete disciplines. Discrete disciplines do not propagate up the hierarchy. This can result in more connect modules being inserted lower down into discrete sections of the design hierarchy for added accuracy.

Examples:

In the example shown in Figure 8-4, `NetA`, `NetB`, `NetC`, and `NetD` are undeclared interconnects.

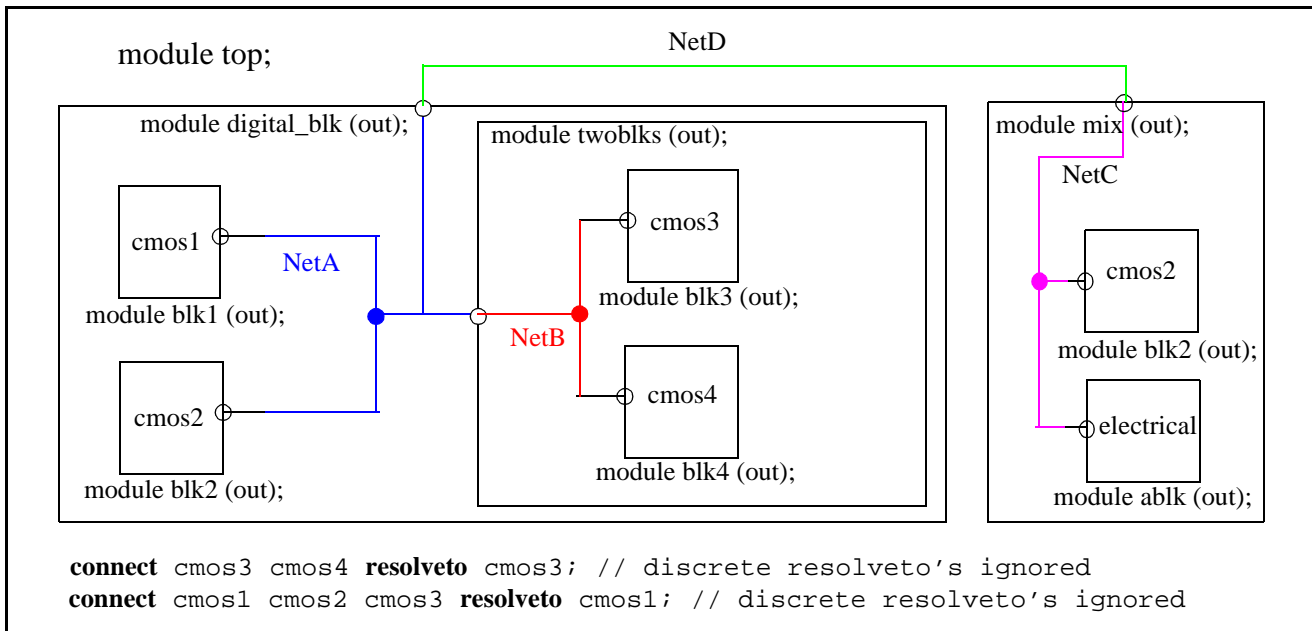


Figure 8-4 Discipline resolution mode: detail

Using the detail mode of discipline resolution for this example results in the following:

- *Continuous up:* NetC resolves to electrical based on continuous (electrical) winning over discrete (cmos2).
- *Continuous up:* NetD resolves to electrical based on continuous (electrical) winning over undeclared.
- *Continuous down:* NetA resolves to electrical based on continuous (electrical) winning over undeclared.
- *Continuous down:* NetB resolves to electrical based on continuous (electrical) winning over undeclared.

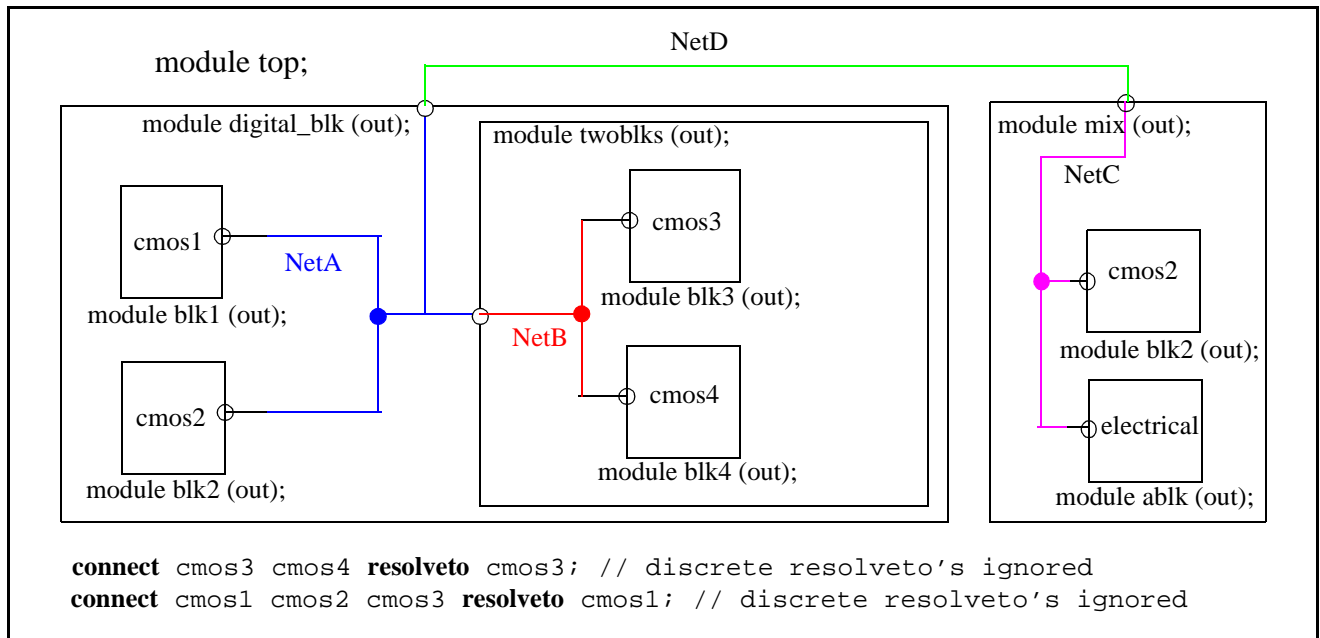
Note: The specified **resolveto** connect statements are ignored in this mode unless coercion (see Section 8.8.1) is used.

8.4.4.3 Coercing discipline resolution

Connect module insertion can be affected by *coercion* i.e., declaring disciplines for the interconnect in the hierarchy. If an interconnect is assigned a discipline, that discipline shall be used unless the **resolveto** connect statement overrides the discipline.

Examples:

The example in Figure 8-5 shows several effects of coercion on auto-insertion.

**Figure 8-5 Coercion effects on auto- insertion**

Case1: NetB is declared as cmos3 (the others are undeclared)

```
cmos3 top.digital_blk.twoblks.NetB
```

discipline resolution basic: Same as without coercion.

discipline resolution detail: NetB stays cmos3; NetA, NetC, and NetD become electrical.

Case2: NetA is declared as cmos1 (the others are undeclared)

discipline resolution basic: NetA stays cmos1, NetB is assigned cmos3, and NetC and NetD become electrical.

discipline resolution detail: Same as basic mode.

Case3: NetC is declared as cmos2 (the others are undeclared)

discipline resolution basic: NetC stays cmos2, NetB is assigned cmos3, NetA is assigned cmos1, and NetD is assigned cmos1.

discipline resolution detail: Same as basic mode.

8.5 Connect modules

Connect modules are automatically inserted to connect the continuous and discrete disciplines (mixed nets) of the design hierarchy together. The continuous and discrete

disciplines of the ports of the connect modules and their directions are used to determine the circumstances in which the module can be automatically inserted.

The connect module is a special form of a module; its definition is shown in Syntax 8-5.

```
connectmodule_declaration ::=
    connectmodule module_identifier ( connectmod_port , connectmod_port ) ;
    [ module_items ]
    endmodule
connectmod_port ::=
    connectmod_port_identifier
```

Syntax 8-5—Syntax for connect modules

8.6 Connect module descriptions

The disciplines of mixed nets are determined prior to the connect module insertion phase of elaboration. Connect module declarations with matching port discipline declarations and directions are instantiated to connect the continuous and discrete domains of the mixed net.

The port disciplines define the default type of disciplines which shall be bridged by the connect module. The directional qualifiers of the discrete port determine the default scenarios where the module can be instantiated. The following combinations of directional qualifiers are supported for the continuous and discrete disciplines of a connect module:

continuous	discrete
input	output
output	input
inout	inout

Examples:

Example 1

```
connectmodule d2a(in, out);
    input in;
    output out;
    logic in;
    electrical out;
    // insert connect module behavioral here
endmodule
```

can bridge a mixed input port whose upper connection is compatible with discipline `logic` and whose lower connection is compatible with `electrical`, or a mixed output port whose upper connection is compatible with discipline `electrical` and whose lower connection is compatible with `logic`.

Example 2

```
connectmodule a2d(out, in);
    output out;
    input in;
    logic out;
    electrical in;
    // insert connect module behavioral here
endmodule
```

can bridge a mixed output port whose upper connection is compatible with discipline `logic` and whose lower connection is compatible with `electrical`, or a mixed input port whose upper connection is compatible with discipline `electrical` and whose lower connection is compatible with `logic`.

Example 3

```
connectmodule bidir(out, in);
    inout out;
    inout in;
    logic out;
    electrical in;
    // insert connect module behavioral here
endmodule
```

can bridge any mixed port whose one connection is compatible with discipline `logic` and whose connection is compatible with `electrical`.

8.7 Connect specification statements

Any number of connect modules can be defined. The designer can choose and specialize those in the design via the connect specification statements. The connect specification statements allow the designer to define:

- specification of which connect module is used, including parameterization, for bridging given discrete and continuous disciplines
- overrides for the connect module default disciplines and port directions
- resolution of incompatible disciplines

The syntax for connect specifications is shown in Syntax 8-6.

```
connect_specification ::=  
    connectrules connectrule_identifier;  
    {connect_spec_item }  
    endconnectrules  
connect_spec_item ::=  
    connect_insertion  
    | connect_resolution
```

Syntax 8-6—Syntax for connect specification statements

The two forms of the connect specification statements and their syntaxes are detailed in the following subsections.

8.7.1 Connect module auto-insertion statement

The connect module insertion statement declares which connect modules are automatically inserted when mixed nets of the appropriate types are encountered, as shown in Syntax 8-7.

This specifies the connect module *connect_module_identifier* is used to determine the mixed nets of the type used in the declaration of the connect module.

There can be multiple connect module declarations of a given (**discrete - continuous**) discipline pair and the connect module specification statement specifies which is to be used in the auto-insertion process. In addition, parameters of the connect module declaration can be specified via the *connect_attributes*.

```

connect_insertion ::=
    connect connect_module_identifier connect_attributes
    [ [ direction ] discipline_identifier , [ direction ] discipline_identifier ] ;

connect_attributes ::=
    [ connect_mode ] [ #( attribute_list ) ]

connect_mode ::=
    merged
    | split

attribute_list ::=
    attribute
    | attribute_list , attribute

attribute ::=
    .parameter_identifier ( expression )

direction ::=
    input
    | output
    | inout

discipline_list ::=
    discipline_identifier
    | discipline_list , discipline_identifier

```

Syntax 8-7—Syntax for connect configuration statements

Connect modules can be reused for different, but compatible disciplines by specifying different discipline combinations in which the connect module can be used. The form is

```
connect connect_module_identifier connect_attributes discipline_identifier , discipline_identifier ;
```

where the specified disciplines shall be compatible for both the continuous and discrete disciplines of the given connect module.

It is also possible to override the port directions of the connect module, which allows a module to be used both as a unidirectional and bidirectional connect module. This override also aids library based designs by allowing the user to specify the connect rules, rather than having to search the entire library. The form is

```
connect connect_module_identifier connect_attributes direction discipline_identifier ,
    direction discipline_identifier ;
```

where the specified disciplines shall be compatible for both the continuous and discrete disciplines of the given connect module and the specified directions are used to define the type of connect module.

8.7.2 Discipline resolution connect statement

The discipline resolution connect statement specifies a single discipline to use during the discipline resolution process when multiple nets with compatible discipline are part of the same mixed net, as shown in Syntax 8-8.

```
connect_resolution ::=
    connect discipline_list resolveto discipline_identifier ;
discipline_list ::=
    discipline_identifier
    | discipline_list , discipline_identifier
```

Syntax 8-8—Syntax for connect configuration resolveto statements

where *discipline_list* is the list of compatible disciplines and *discipline_identifier* is the discipline to be used.

8.7.2.1 Connect Rule Resolution Mechanism

When there is an exact match for the set of disciplines specified as part of the *discipline_list*, the resolved discipline would be as per the rule specified in the exact match. When more than one specified rule applies to a given scenario a "Warning" message shall be issued by the simulator and the first match would be used.

When there is no exact fit, then the resolved discipline would be based on the subset of the rules specified. If there is more than one subset matching a set of disciplines, the simulator shall give a "Warning" message and apply the first subset rule that satisfies the current scenario.

The resolved discipline need not be one of the disciplines specified in the discipline list.

The *connect-resolveto* shall not be used as a mechanism to set the disciplines of simulator primitives but used only for discipline resolution.

Example 1:

```
connect x,y,a resolveto a;
connect x,y resolveto x;
```

For the above set of connect rule specifications,

disciplines *x,y* would resolve to discipline *x*.

disciplines *x,y,a* would resolve to discipline *a*.

disciplines *y,a* would resolve to discipline *a*.

Example 2:

```
connect x,y,a resolveto y;  
connect x,y,a resolveto a;  
connect x,y,b resolveto b;
```

For the above set of connect rule specifications,

disciplines *x,y* would resolve to discipline *y* with a warning.

disciplines *x,y,a* would resolve to discipline *y* with a warning.

disciplines *y,b* would resolve to *b*.

8.7.3 Parameter passing attribute

An attribute method can be used with the connect statement to specify parameter values to pass into the Verilog-AMS HDL connect module and override the default values. Any parameters declared in the connect module can be specified.

Examples:

```
connect a2d_035u #(.tt(3.5n), .vcc(3.3));
```

Here each parameter is listed with the new value to be used for that parameter.

8.7.4 connect_mode

This can be used to specify additional segregation of connect modules at each level of the hierarchy. Setting *connect_mode* to **split** or **merged** defines whether all ports of a common discrete discipline and port direction share an connect module or have individual connect modules.

Examples:

```
connect a2d_035u split #(.tt(3.5n), .vcc(3.3));
```

Here each digital port has a separate connect module.

8.8 Automatic insertion of connect modules

Automatic insertion of connect modules is performed when signals and ports with continuous time domain and discrete time domain disciplines are connected together. The connect module defines the conversion between these different disciplines.

An instance of the connect module shall be inserted across any mixed port that matches the rule specified by a **connect** statement. Rules for matching connect statements with ports take into account the port direction (see Section 8.8.1) and the disciplines of the signals connected to the port.

Each `connect` statement designates a module to be a connect module. When two disciplines are specified in a connect statement, one shall be discrete and the other continuous.

Examples:

```

module dig_inv(in, out);
    input in;
    output out;
    reg out;
    logic in, out;

    always begin
        out = #10 ~in;
    end
endmodule

module analog_inv(in, out);
    input in;
    output out;
    electrical in, out;
    parameter real vth = 2.5;
    real outval;

    analog begin
        if (V(in) > vth)
            outval = 0;
        else
            outval = 5 ;
        V(out) <+ transition(outval);
    end
endmodule

module ring;

    dig_inv d1 (n1, n2);
    dig_inv d2 (n2, n3);
    analog_inv a3 (n3, n1);

endmodule

connectmodule elect_to_logic(el,cm);
    input el;
    output cm;
    reg cm;
    electrical el;
    logic cm;

    always
        @(cross(V(el) - 2.5, 1))
            cm = 1;

    always
        @(cross(V(el) - 2.5, -1))
            cm = 0;

```

```

endmodule

connectmodule logic_to_elect(cm,el);
    input cm;
    output el;
    logic cm;
    electrical el;

    analog
        V(el) <+ transition((cm == 1) ? 5.0 : 0.0);

endmodule

connectrules mixedsignal;
    connect elect_to_logic;
    connect logic_to_elect;
endconnectrules

```

Here two modules, `elect_to_logic` and `logic_to_elect`, are specified as the connect modules to be automatically inserted whenever a signal and a module port of disciplines `electrical` and `logic` are connected.

Module `elect_to_logic` converts signals on port `out` of instance `a3` to port `in` of instance `d1`. Module `logic_to_elect` converts the signal on port `out` of instance `d2` to port `in` of instance `a3`.

8.8.1 Connect module selection

The selection of a connect module for automatic insertion depends upon the disciplines of nets connected together at ports. It is, therefore, a post elaboration operation since the signal connected to a port is only known when the module in which the port is declared has been instantiated.

Auto-insertion of connect modules is done hierarchically. The connect modules are inserted based on the net disciplines and ports at each level of the hierarchy. The *connect_mode* **split** and **merged** are applied at each level of the hierarchy. This insertion supports the ability to coerce the placement of connect modules by declaring the disciplines of interconnect.

Example:

Figure 8-6 shows an example of auto-insertion with coercion.

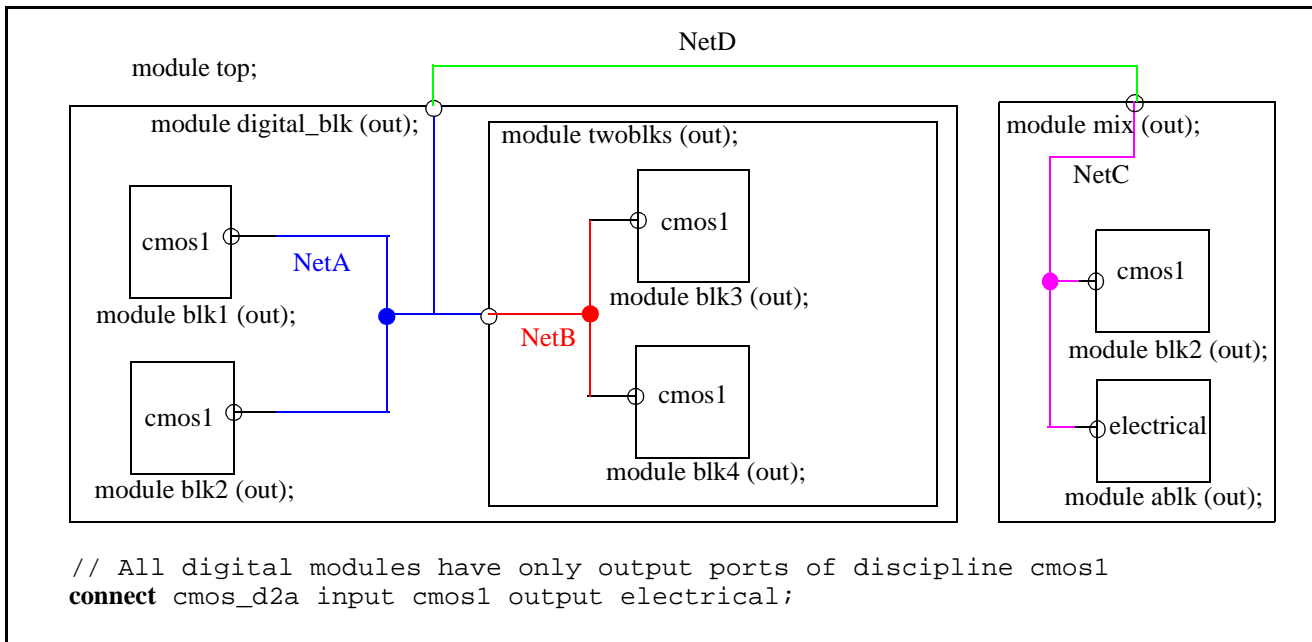


Figure 8-6 Auto-insertion with coercion

Case1: All interconnects are undeclared

discipline resolution basic:

merged: d2a at top.mix.blk2 and d2a at top.digital_blk (two connect modules).

split: Same as *merged*.

discipline resolution detail:

merged: d2a at top.mix.blk2, d2a at top.digital_blk.(blk1-blk2), and d2a at top.digital_blk.twoblks (three connect modules).

split: d2a at each of the five cmos1 blocks.

Case2: If NetB is declared as cmos1 and the remaining interconnect is undeclared

discipline resolution basic:

merged: d2a at top.mix.blk2 and d2a at top.digital_blk (two connect modules).

split: Same as *merged*.

discipline resolution detail:

merged: d2a at top.mix.blk2, d2a at top.digital_blk.(blk1-blk2), and d2a at top.digital_blk.twoblks (three connect modules).

split: d2a at top.mix.blk2, d2a at top.digital_blk.blk1, d2a at top.digital_blk.blk2, and d2a at top.digital_blk.twoblks (four connect modules).

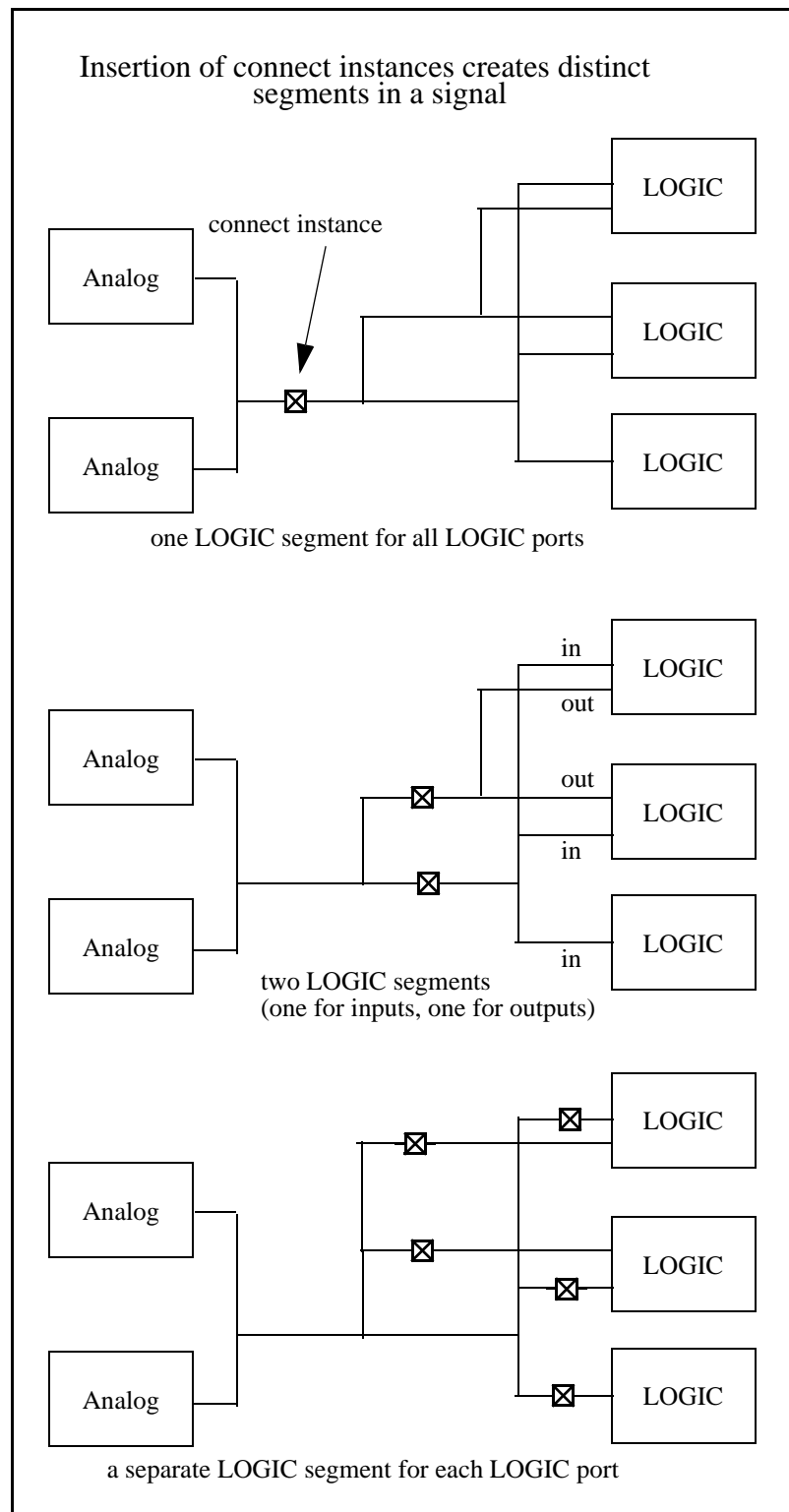
8.8.2 Signal segmentation

Once a connect module has been selected it can not be inserted until it can be determined whether there should be one connect module per port or one connect module for all the ports on the net of a signal which match a given `connect` statement. Inserting multiple copies of the same connect module on one signal (i.e., between the signal and the multiple ports) has the effect of creating distinct segments of the signal with the same discipline at that level of the hierarchy.

This segmentation of the signal which connects ports is only performed in the case of digital ports (i.e., ports with discrete-time domain or digital discipline). For analog (or continuous-time domain) disciplines, it is not desirable to segment the signal between the ports; i.e, there shall never be more than one analog node representing a signal. However, it can be desirable for the simulator's internal representation of the signal to consist of various separate digital segments, each with its own connect module.

Examples:

Figure 8-7 shows how to model the loading effect of each individual digital port on the analog node.

**Figure 8-7 Signal segmentation by connect modules**

8.8.3 **connect_mode** parameter

This parameter can be used in the `connect` statement to direct the segmentation of the signal at each level of the hierarchy, which can occur while inserting a connect module. It can be one of two predefined values, **split** or **merged**. The default is **merged**.

The `connect_mode` indicates how input, output, or inout ports of the given discipline shall be combined for the purpose of inserting connect modules. It is applied when there is more than one port of discrete discipline on a net of a signal where the `connect` statement applies.

8.8.3.1 **merged**

This instructs the simulator to try to group all ports (whether they are input, output, or inout) and to use just one connector module, provided the module is the same.

Example:

Figure 8-8 illustrates the effect of the **merged** attribute.

Connection of the `electrical` signal to the `t1` inout ports and `t1` input ports results in a single connector module, `bidir`, being inserted between the ports and the `electrical` signal. The `t1` output ports are merged, but with a different connect module; i.e., there is one connector module inserted between the `electrical` signal and all of the `t1` output ports.

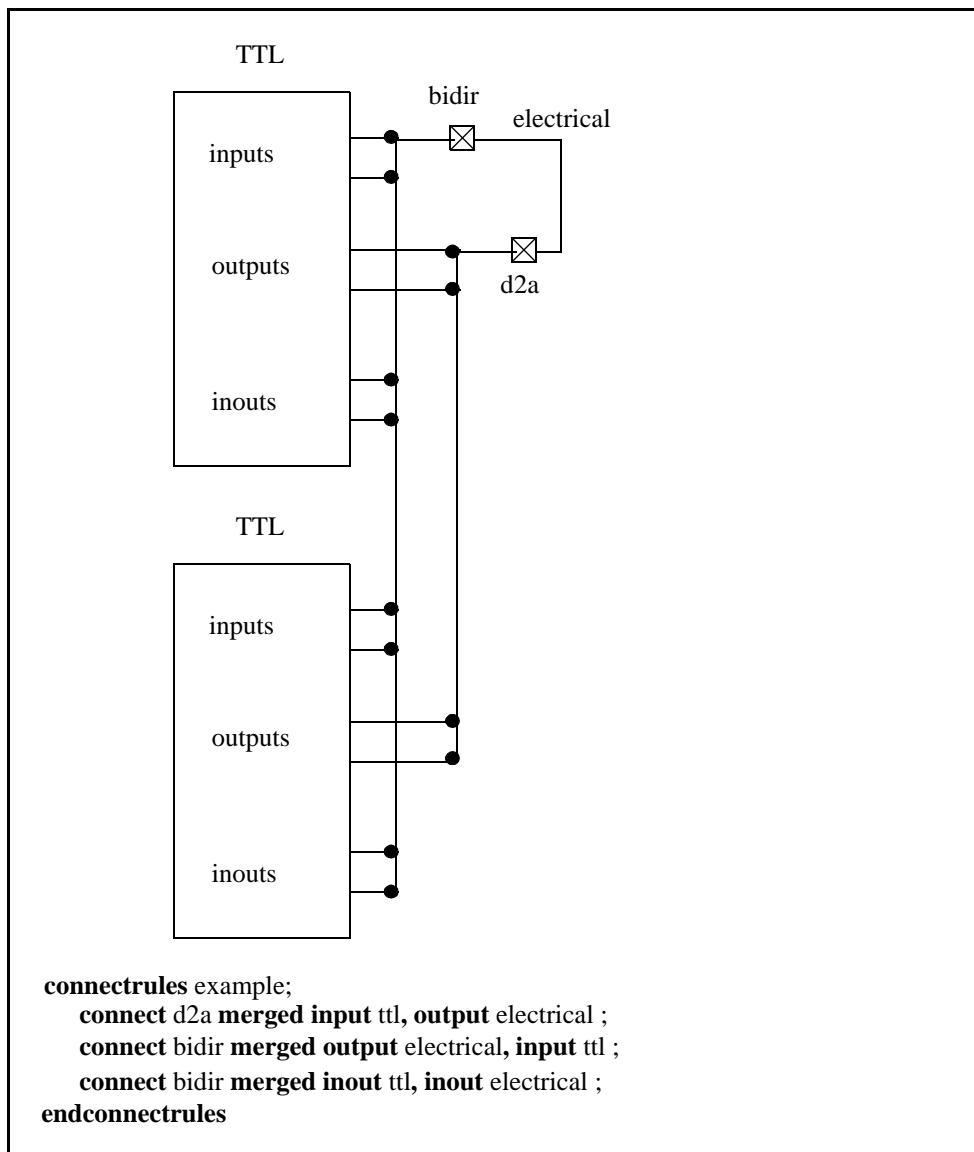


Figure 8-8 Connector insertion using merged

8.8.3.2 split

If more than one input port is connected at a net of a signal, using `split` forces there to be one connect module for each port which converts between the net discipline and the port discipline. In this way, the net connecting to the ports is segmented by the insertion of one connect module for each port.

Examples:

Example 1

```
connect elect_to_logic split;
```

This `connect` statement specifies the module `elect_to_logic` shall be split across the discrete module ports:

- if an input port has `logic` discipline and the signal connecting to the port has `electrical` discipline, or
- if an output port has `electrical` discipline and the signal connecting to the port has `logic` discipline.

Example 2

In Figure 8-9, the connections of an `electrical` signal to `t1` output ports results in a distinct instance of the `d2a` connect module being inserted for each output port. This is mandated by the `split` parameter.

Connection of the `electrical` signal to `t1` input ports results in a single instance of the `a2d` connect module being inserted between the `electrical` signal and all the `t1` input ports. This is mandated by `merged` parameter. This behavior is also seen for the `t1` inout ports where the `merged` parameter is used.

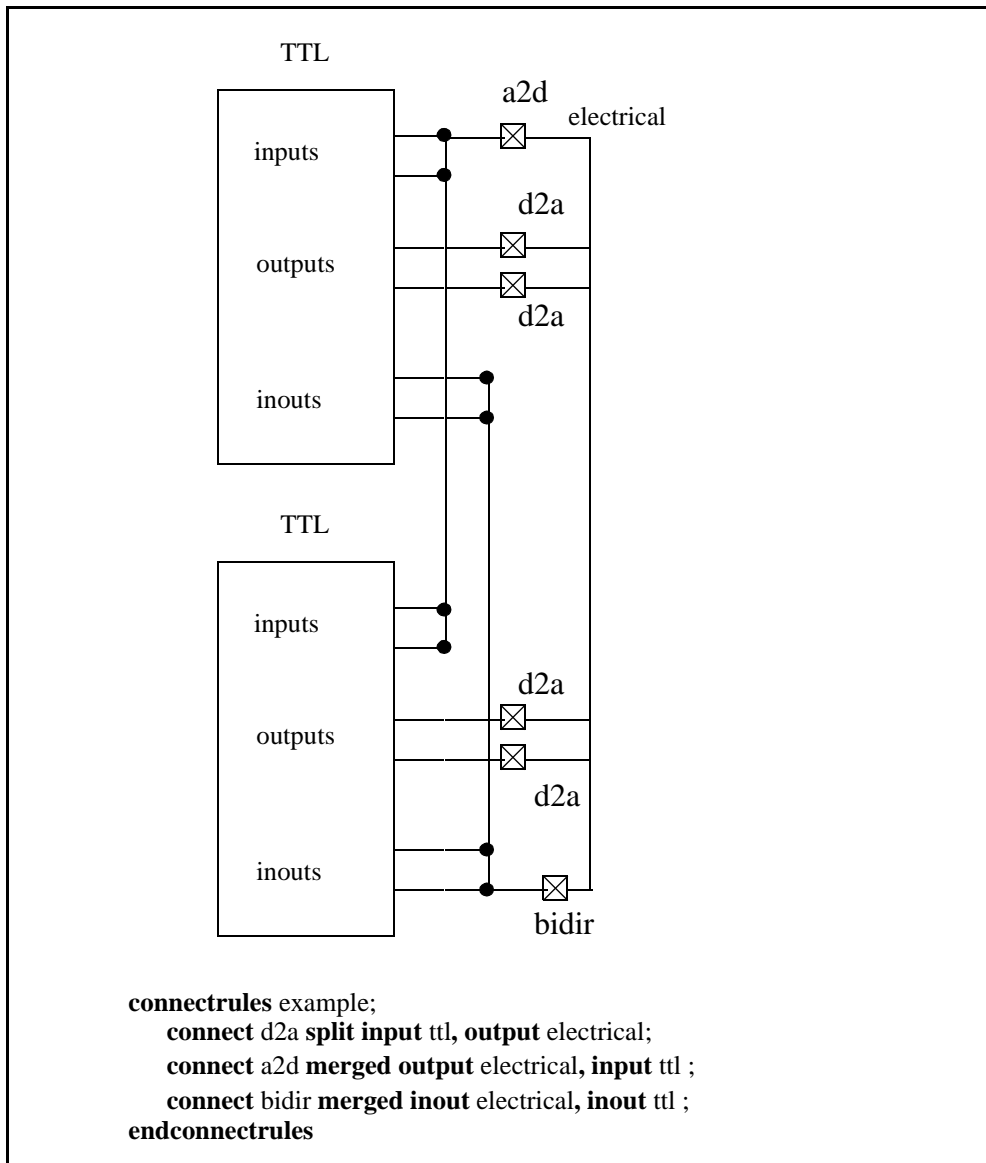


Figure 8-9 Connect module insertion with signal segmentation

Example 3

This example:

```
connect cmosA2d split #(.r(30k) input electrical, output cmos02u;
```

performs three functions:

1. connects an instance of `cmosA2d` module between a signal with `electrical` discipline and the input port with `cmos02u` discipline, or an output port with `electrical` discipline and the signal with `cmos02u` discipline;

2. sets the value of the parameter `r` to 30k; and
3. uses one module instance for each input port.

If there are many output ports where this rule applies, by definition there is no segmentation of the signal between these ports, since the ports have discipline `electrical` (an analog discipline).

Example 4

This last example:

```
connect cmosA2d merged #(.r(15k) input electrical, output cmos04u;
```

does three things:

1. connects an instance of `cmosA2d` module between a signal with electrical discipline and an input port with `cmos04u` discipline, or an output port with `electrical` discipline and a signal with `cmos4u` discipline;
2. sets the value of the parameter `r` to 15k; and
3. uses one module instance regardless of the number of ports.

8.8.4 Rules for driver-receiver segregation and connect module selection and insertion

Driver-receiver segregation and connect module insertion is a post elaboration operation. It depends on a complete hierarchical examination of each signal in the design, i.e., an examination of the signal in all the contexts through which it passes. If the complete hierarchy of a signal is digital, i.e., the signal has a digital discipline in all contexts through which it passes, it is a *digital signal* rather than a mixed signal. Similarly, if the complete hierarchy of a signal is analog, it is an *analog signal* rather than a mixed signal. Rules for driver-receiver segregation and connect module insertion apply only to *mixed signals*, i.e., signals which have an analog discipline in one or more of the contexts through which they pass and a digital discipline in one or more of the contexts. In this case, *context* refers to the appearance of a signal in a particular module instance.

For a particular signal, a module instance has a digital context if the signal has a digital discipline in that module or an analog context if the signal has an analog discipline. The appearance of a signal in a particular context is referred to as a *segment* of the signal. In general, a *signal* in a fully elaborated design consists of various segments, some of which can be analog and some of which can be digital.

A *port* represents a connection between two net segments of a signal. The context of one of the net segments is an instantiated module and the context of the other is the module which instantiates it. The segment in the instantiated module is called the *lower* or *formal connection* and the segment in the instantiating module is the *upper* or *actual connection*. A connection element is selected for each port where one connection is analog and the other digital.

The following rules govern driver-receiver segregation and connect module selection. These rules apply only to mixed signals.

1. A mixed signal is represented in the analog domain by a single node, regardless of how its analog contexts are distributed hierarchically.
2. Digital drivers of mixed signals are segregated from receivers so the digital drivers contribute to the analog state of the signal and the analog state determines the value seen by the receivers.
3. A connection shall be selected for a port only if one of the connections to the port is digital and the other is analog. In this case, the port shall match one (and only one) connect statement. The module named in the connect statement is the one which shall be selected for the port.

Once connect modules have been selected, they are inserted according to the `connect_mode` parameter in the pertinent connect statements. These rules apply to connect module insertion:

1. The connect mode of a port for which a connect module has been selected shall be determined by the value of the `connect_mode` parameter of the connect statement which was used to select the connect module.
2. The connect module for a port shall be instantiated in the context of the ports upper connection.
3. All ports connecting to the same signal (upper connection), sharing the same connect module, and having `merged` parameter shall share a single instance of the selected connect module.
4. All other ports shall have an instance of the selected connect module, i.e., one connect module instance per port.

8.8.5 Instance names for auto-inserted instances

Parameters of auto-inserted connect instances can be set on an instance-by-instance basis with the use of the `defparam` statement. This requires predictable instance names for the auto-inserted modules.

The following naming scheme is employed to unambiguously distinguish the connector modules for the case of auto-inserted instances.

1. **merged**
In the merged case, one or more ports have a given discipline at their bottom connection, call it `BottomDiscipline`, and a common signal, call it `SigName`, of another discipline at their top connection. A single connect module, call it `ModuleName`, is placed between the top signal and the bottom signals. In this case,

the instance name of the connect module is derived from the signal name, module name, and the bottom discipline:

`SigName__ModuleName__BottomDiscipline`

2. **split**

In the split case, one or more ports have a given discipline at their bottom connection and a common signal of another discipline, call it `TopDiscipline`, at their top connection. One module instance is instantiated for each such port. In this case, the instance name of the connect module is

`SigName__InstName__PortName`

where `InstName` and `PortName` are the local instance name of the port and its instance respectively.

Note: The `__` between the elements of these generated instance names is a double underscore.

8.8.5.1 Port names for Verilog built-in primitives

In the cases of instances of modules and instances of UDPs, port names are well defined. In these cases the port name is the name of the signal at the lower connection of the port. In the case of built-in digital primitives, however, IEEE 1364-2001 does not define port names. In order to support the unique naming of auto inserted connect modules and the ability to override the parameters of those connect modules, built-in digital primitives ports will be provided with predictable names. These names are only for the purpose of naming the connect modules and do not define actual port names. These port names may not be used to instantiate or to do access of these primitives.

The following naming conventions shall be used when generating connect module instance names that are connected to built-in digital primitives.

1. For N-input gates (and, nand, nor, or, xnor, xor) the output will be named `out`, and the inputs reading from left to right will be `in1`, `in2`, `in3`, and so forth.
2. For N-output gates (buf, not) The input will be named `in`, and the outputs reading from left to right will be named `out1`, `out2`, `out3`, and so forth.
3. For 3 port MOS switches (nmos, pmos, rnmos, rpmos) the ports reading from left to right will be named `source`, `drain`, `gate`.
4. For 4 port MOS switches (cmos, rcmos) the ports reading from left to right will be named `source`, `drain`, `ngate`, `pgate`.
5. For bidirectional pass switches (tran, tranif1, tranif0, rtran, rtranif1, rtranif0) the ports reading from left to right will be named `source`, `drain`, `gate`.
6. For single port primitives (pullup, pulldown) the port will be named `out`.

8.9 Driver-receiver segregation

If the hierarchical segments of a signal are all digital or all analog, the signal is not a mixed signal and the internal representation of the signal does not differ from that of a purely digital or an analog signal.

If the signal has both analog and digital segments in its hierarchy, it is a mixed signal. In this case, the appropriate conversion elements are inserted, either manually or automatically, based on the following rules.

- All the analog segments of a mixed signal are representations of a single analog node.
- Each of the non-contiguous digital segments of a signal shall be represented internally as a separate digital signal, with its own state.
- Each non-contiguous digital segment shall be segregated into the collection of drivers of the segment and the collection of receivers of the segment.

In the digital domain, signals can have drivers and receivers. A driver makes a contribution to the state of the signal. A receiver accesses, or reads, the state of the signal. In a pure digital net, i.e., one without an analog segment, the simulation kernel resolves the values of the drivers of a signal and it propagates the new value to the receivers by means of an event when there is a change in state.

In the case of a mixed net, i.e., one with digital segments and an analog segment, it can be useful to propagate the change to the analog simulation kernel, which can then detect a threshold crossing, and then propagate the change in state back to the digital kernel. This, among other things, allows the simulation to account for rise and fall times caused by analog parasitics.

Within digital segments of a mixed-signal net, drivers and receivers of ordinary modules shall be segregated, so transitions are not propagated directly from drivers to receivers, but propagate through the analog domain instead. In this case, the drivers and receivers of connect modules shall be oppositely segregated; i.e., the connect module drivers shall be grouped with the ordinary module receivers and the ordinary module drivers shall be grouped with the connect module receivers.

Thus, digital transitions are propagated from drivers to receivers by way of analog (through using connect module instances). Figure 8-10 and Figure 8-11 show driver-receiver segregation in modules having bidirectional and unidirectional ports, respectively.

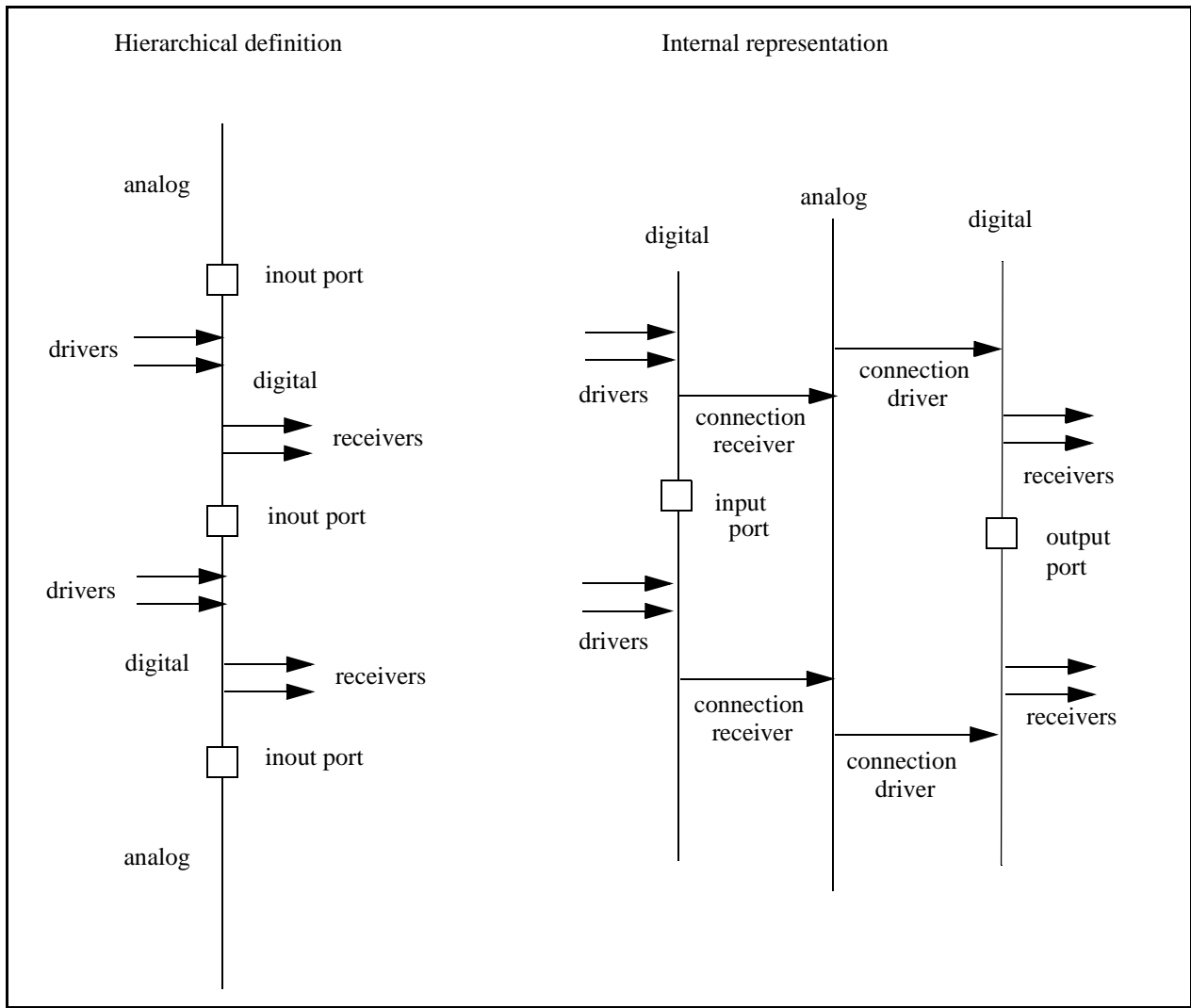


Figure 8-10 Driver-receiver segregation in modules with bidirectional ports

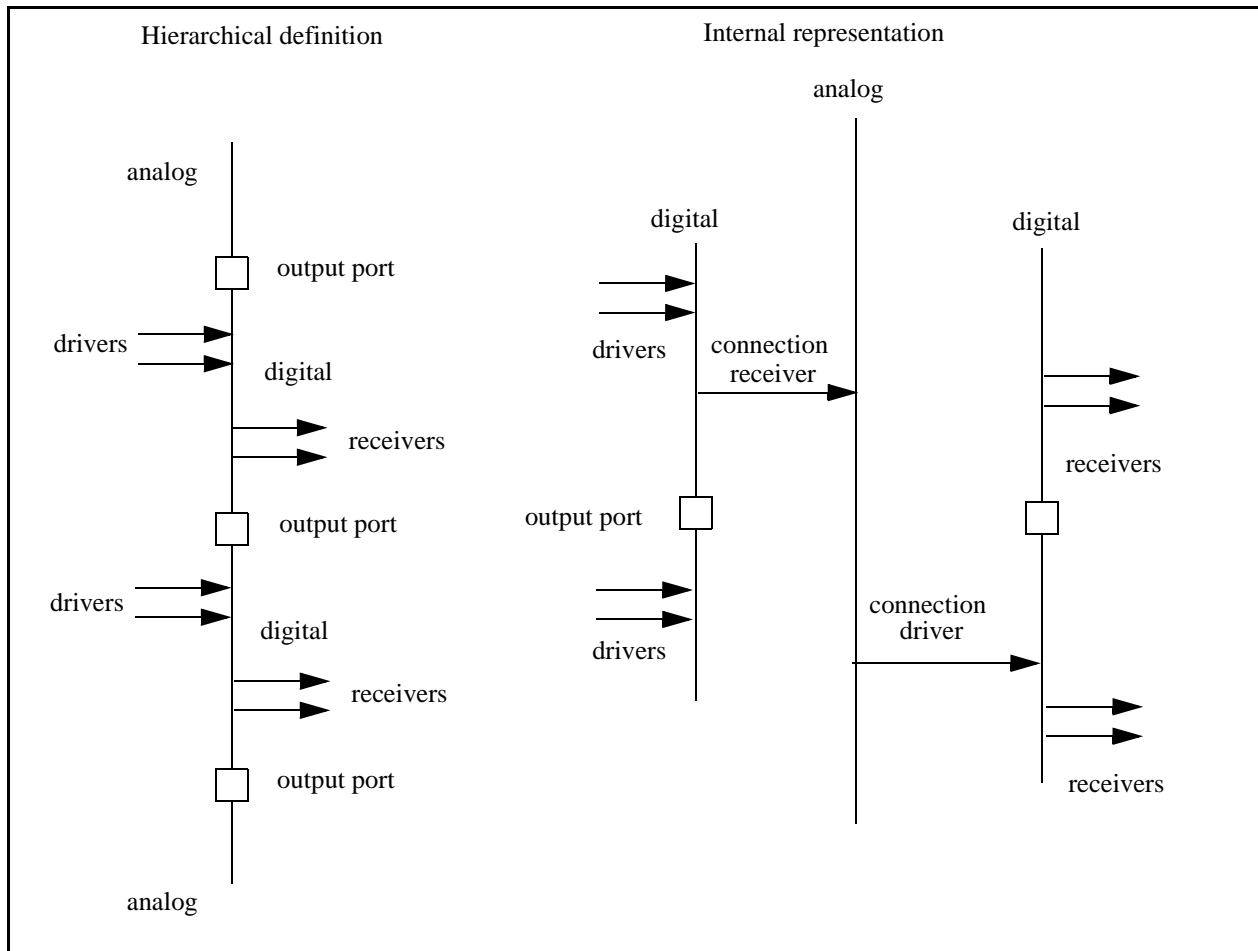


Figure 8-11 Driver-receiver segregation in modules with unidirectional ports

8.10 Driver access and net resolution

Access to individual drivers and net resolution is necessary for accurate implementation of connect modules (see Section 8.5). A *driver* of a signal is a process which assigns a value to the signal, or a connection of the signal to an output port of a module instance or simulation primitive. The driver access functions described here only access drivers found in ordinary modules and not to those found in connect modules. Driver access functions can only be called from connect modules.

A signal can have any number of drivers; for each driver the current status, value, and strength can be accessed.

8.10.1 **\$driver_count**

\$driver_count returns an integer representing the number of drivers associated with the signal in question. The syntax is shown in Syntax 8-9.

```
driver_count_function ::=
    $driver_count ( signal_name )
```

Syntax 8-9—Syntax for \$driver_count

The drivers are arbitrarily numbered from 0 to $N-1$, where N is the total number of ordinary drivers contributing to the signal value. For example, if this function returns a value 5 then the signal has five drivers numbered from 0 to 4.

8.10.2 **\$driver_state**

driver_state returns the current value contribution of a specific driver to the state of the signal. The syntax is shown in Syntax 8-10.

```
driver_state_function ::=
    $driver_state ( signal_name , driver_index )
```

Syntax 8-10—Syntax for \$driver_state

driver_index is an integer value between 0 and $N-1$, where N is the total number of drivers contributing to the signal value. The state value is returned as 0, 1, x, or z.

8.10.3 **\$driver_strength**

driver_strength returns the current strength contribution of a specific driver to the strength of the signal. The syntax is shown in Syntax 8-11.

```
driver_strength_function ::=
    $driver_strength ( signal_name , driver_index )
```

Syntax 8-11—Syntax for \$driver_strength

driver_index is an integer value between 0 and $N-1$, where N is the total number of drivers contributing to the signal value. The strength value is returned as two strengths, Bits 5-3 for strength0 and Bits 2-0 for strength1 (see *IEEE 1364-1995 Verilog HDL*, sections 7.10 and 7.11).

If the value returned is 0 or 1, strength0 returns the high-end of the strength range and strength1 returns the low-end of the strength range. Otherwise, the strengths of both strength0 and strength1 is defined as shown in Figure 8-12 below.

strength0									strength1								
Bits	7 Su0	6 St0	5 Pu0	4 La0	3 We0	2 Me0	1 Sm0	0 HiZ0	0 HiZ1	1 Sm1	2 Me1	3 We1	4 La1	5 Pu1	6 St1	7 Su1	Bits
B5	1	1	1	1	0	0	0	0	0	0	0	0	1	1	1	1	B2
B4	1	1	0	0	1	1	0	0	0	0	1	1	0	0	1	1	B1
B3	1	0	1	0	1	0	1	0	0	1	0	1	0	1	0	1	B0

Figure 8-12 Strength value mapping

8.10.4 driver_update

The status of drivers for a given signal can be monitored with the event detection keyword **driver_update**. It can be used in conjunction with the event detection operator @ to detect updates to any of the drivers of the signal.

Examples:

```
always @(driver_update clock)
    statement;
```

causes the `statement` to execute any time a driver of the signal `clock` is updated. Here, an update is defined as the addition of a new pending value to the driver. This is true whether or not there is a change in the resolved value of the signal.

8.10.5 Receiver net resolution

As a result of driver receiver segregation, the drivers and receivers are separated so that any analog connected to a mixed net has the opportunity to influence the value driving the digital receivers. Since a single digital port is used in the connect module, the user must specify the value that the receivers will see. By not specifying the receiver value directly in the connect module driver, receiver segregation will be ignored, which is the default case. This assignment of the receiver value is done via the **assign** statement in which the digital port will be used to read the driver values as well as to set the receiver value.

1. The default is equivalent of **assign** `d_receivers = d_drivers ;`

Where the value passed to the receivers through driver receiver segregation is the value being driven without delay or any impact from analog connections to the net. This is essentially bypassing driver receiver segregation.

2. Anything else is done explicitly, such as:

```
reg out; // value of out determined in CM, see example
// Section 8.10.6
assign d = out;
```

In this case, the digital port of the connect module will drive the receivers with a value determined in the connect module. This value may potentially be different from the value of the drivers of the connect module digital port.

8.10.6 Connect module example using driver access functions

Using the example shown in Figure 8-13, a connect module can be created using driver access functions to accurately model the effects of multiple drivers on an interface.

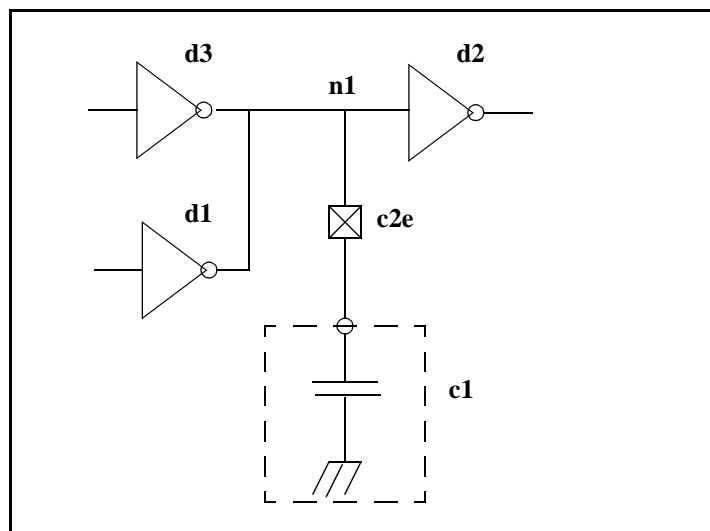


Figure 8-13 Driver-receiver segregation connect module example

The connect module below takes advantage of much of the mixed-signal language including driver access functions. This module effectively adds another parallel resistor from output to ground whenever a digital output connected to the net goes high, and another parallel resistor from output to rail (*supply*) whenever a digital output connected to the net goes low. If this is used as the connect module in Figure 8-13, not only is the delay from digital outputs to the digital input a function of the value of the capacitor, for a given capacitance it takes approximately half the time (since two gates are driving the signal rather than one).

```
connectmodule c2e(d,a);
input d;
output a;

logic d;
electrical a, rail, gnd;
```

```

reg out;
ground gnd;
branch (rail,a)pull_up;
branch (a,gnd)pull_down;
branch (rail,gnd)power;
parameter real impedance0 = 120.0;
parameter real impedance1 = 100.0;
parameter real impedanceOff = 1e6;
parameter real vt_hi = 3.5;
parameter real vt_lo = 1.5;
parameter real supply = 5.0;
integer i, num_ones, num_zeros;

assign d=out;
initial begin
    num_ones = 0;
    num_zeros = 0;
end

always @(driver_update(d)) begin
    num_ones = 0;
    num_zeros = 0;
    for ( i = 0; i < $driver_count(d); i=i+1)
        if ( $driver_state(d,i) == 1 )
            num_ones = num_ones + 1;
        else
            num_zeros = num_zeros + 1;
    end

    always @(cross(V(a) - vt_hi, -1) or cross(V(a) - vt_lo, +1))
        out = 1'bx;
    always @(cross(V(a) - vt_hi, +1))
        out = 1'b1;
    always @(cross(V(a) - vt_lo, -1))
        out = 1'b0;

    analog begin
        // Approximately one impedance1 resistor to rail per high output
        // connected to the digital net
        V(pull_up) <+ 1/((1/impedance1)*num_ones+(1/impedanceOff)) *
            I(pull_up);
        // Approximately one impedance0 resistor to ground per low output
        // connected to the digital net
        V(pull_down) <+ 1/((1/impedance0)*num_zeros+(1/impedanceOff)) *
            I(pull_down);
        V(power) <+ supply;
    end
end
endmodule

```

8.11 Supplementary driver access functions

The following driver access functions are provided for access to digital events which have been scheduled onto a driver but might not have matured by the current simulation time.

These functions can be used to create analog waveforms which cross a specified threshold at the same time the digital event matures, thus providing accurate registration of analog and digital representations of a signal. This assumes there is at least as long a delay in the maturation of the digital signal as the required rise/fall times of the analog waveform.

Note: Because the scheduled digital events can be scheduled with an insufficient delay or cancelled before they mature, be careful when using these functions.

8.11.1 \$driver_delay

\$driver_delay returns the delay, from current simulation time, after which the pending state or strength becomes active. If there is no pending value on a signal, it returns the value minus one (-1.0). The syntax is shown in Syntax 8-12.

```
driver_delay_function ::=  
    $driver_delay ( signal_name , driver_index )
```

Syntax 8-12—Syntax for \$driver_delay

driver_index is an integer value between 0 and $N-1$, where N is the total number of drivers contributing to the signal value. The returned delay value is a real number, which is defined by the ``timescale` for that module where the call has been made. The fractional part arises from the possibility of a driver being updated by an A2D event off the digital timeticks.

8.11.2 \$driver_next_state

\$driver_next_state returns the pending state of the driver, if there is one. If there is no pending state, it returns the current state. The syntax is shown in Syntax 8-13.

```
driver_next_state_function ::=  
    $driver_next_state ( signal_name , driver_index )
```

Syntax 8-13—Syntax for \$driver_next_state

driver_index is an integer value between 0 and $N-1$, where N is the total number of drivers contributing to the signal value. The pending state value is returned as 1'b0, 1'b1, 1'bx, or 1'bz.

8.11.3 **\$driver_next_strength**

\$driver_next_strength returns the strength associated with the pending state of the driver, if there is one. If there is no pending state, it returns the current strength. The syntax is shown in Syntax 8-14.

```
driver_next_strength_function ::=
    $driver_next_strength ( signal_name , driver_index )
```

Syntax 8-14—Syntax for **\$driver_next_strength**

driver_index is an integer value between 0 and $N-1$, where N is the total number of drivers contributing to the signal value. The pending strength value is returned as an integer between 0 and 7.

8.11.4 **\$driver_type**

\$driver_type returns an integer value with its bits set according to the system header file “driver_access.vams” (refer to Annex D for the header file) for the driver specified by the *signal_name* and the *driver_index*. Connect modules for digital to analog conversion can use the returned information to help minimize the difference between the digital event time and the analog crossover when the user swaps between coding styles and performs backannotation¹. A simulator that cannot provide proper information for a given driver type should return 0 (‘DRIVER_UNKNOWN’). All drivers on *wor* and *wand* nets will have a bit set indicating such, and any extra drivers added by the kernel for pull-up or pull-down will be marked as belonging to the kernel. The syntax is shown in Syntax 8-15.

```
driver_type_function ::=
    $driver_type( signal_name , driver_index )
```

Syntax 8-15—Syntax for **\$driver_type**

Digital primitives (like nand and nor gates) should always provide data about their scheduled output changes; i.e., a gate with a 5ns delay should provide 5ns of look-ahead.

1. SDF backannotation will not change which D2A is inserted.

Behavioral code with blocking assigns cannot provide look-ahead, but non-blocking assigns with delays can. However, since the capability is implementation- and configuration-dependent, this function is provided so that the connect module can adapt for a particular instance.

Section 9

Scheduling semantics

This section details the simulation cycles for analog simulation and mixed A/D simulations.

9.1 Introduction

A mixed-signal simulator shall contain an analog solver that complies with the analog simulation cycle described in Section 9.2. This component of the mixed-signal simulator is termed the analog engine. A mixed signal simulator shall also contain a discrete event simulator that complies with the scheduling semantics described in Section 9.4. This component is termed the digital engine.

In a mixed-signal circuit, an “analog macro-process” is a set of continuous nodes that must be solved together because they are joined by analog blocks or analog primitives. A mixed-signal circuit can comprise one or more analog macro-process separated by digital processes.

9.2 Analog simulation cycle

Simulation of a network, or system, starts with an analysis of each node to develop equations which 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.

9.2.1 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 (KFL).

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 shall be zero (0) by definition.

9.2.2 Transient analysis

The equation describing the network is differential and non-linear, 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, as shown in Figure 9-1.

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 Newton-Raphson (NR) method to solve this system.

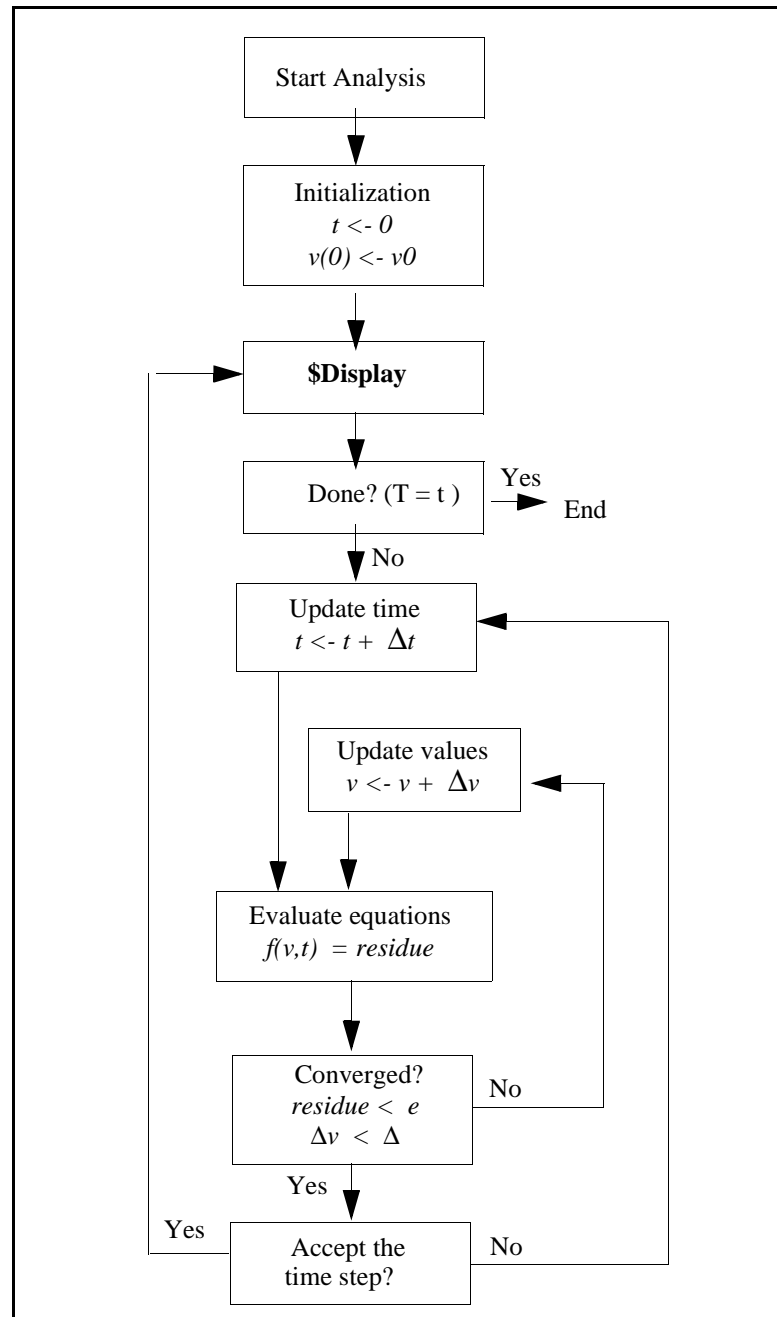


Figure 9-1 Simulation flowchart (transient analysis)

9.2.3 Convergence

In the analog kernel, the behavioral description is evaluated iteratively until the NR method converges. On the first iteration, the signal values used in expressions are approximate and do not satisfy Kirchhoff's Laws.

In fact, the initial values might not be reasonable, so models need to be written so they do something reasonable even when given unreasonable signal values.

For example, the log or square root of a signal value is being computed, 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 the proposed solution on this iteration, $v_n^{(j)}(t)$, shall be close to the proposed solution on the previous iteration, $v_n^{(j-1)}(t)$, 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, which one is used depends on the quantity the signal represents (volts, amps, etc.). The absolute tolerance is important when v_n is converging to zero (0). Without *abstol*, the iteration never converges.

The second criterion ensures Kirchhoff's Flow Law is satisfied:

$$\left| \sum_n f_n^i(v^{(j)}) \right| < reltol(\max(|f_n^i(v^{(j)})|)) + abstol$$

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 convergence is not precluded when v_n or $f_n(v)$ go to zero (0). The relative tolerance can be set once in an options statement to work effectively on any node in the circuit, but the absolute tolerance shall be scaled appropriately for its associated signal. The absolute tolerance shall be the largest signal value which is considered negligible on all the signals where 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.

9.3 Mixed-signal simulation cycle

This section describes the semantics of the initialization and time-sweep phases of a transient analysis in a mixed-signal simulation cycle.

9.3.1 Circuit initialization

The initialization phase of a transient analysis is the process of initializing the circuit state before advancing time.

9.3.2 Synchronization of analog and digital in transient analysis

A Verilog-AMS simulation consists of a number of analog and digital processes communicating via events, shared memory and conservative nodes. Analog processes that share conservative nodes are “solved” jointly and can be viewed as a “macro” process, there may be any number “macro” processes, and it is left up to the implementation whether it solves them in a single matrix, multiple matrices or uses other techniques but it should abide by the accuracy stipulated in the disciplines and analog functions.

9.3.2.1 Concurrency

Most (current) simulators are single-threaded in execution, meaning that although the semantics of Verilog-AMS imply processes are active concurrently, the reality is that they are not. If an implementation is genuinely multi-threaded, it should not evaluate processes that directly share memory concurrently, as there are no data locking semantics in Verilog-AMS.

9.3.2.2 Analog macro process scheduling semantics

The internal evaluation of an analog macro process is described in Section 9.2.2. Once the analog engine has determined its behavior for a given time, it must communicate the results to other processes in the mixed signal simulation through events and shared variables. When an analog macro process is evaluated, the analog engine finds a potential “solution” at a future time (the “acceptance time”), and it stores (but does not communicate) values¹ for all the process’s nodes up to that time. A “wake up” event is scheduled for the acceptance time of the process, and the process is then inactive until it is either woken up or receives an event from another process. If it is woken up by its own “wake up” event, it calculates a new solution point, acceptance time (and so forth) and deactivates. If it is woken up prior to acceptance time by an event that disturbs its current solution, it will cancel its own “wake up” event, accept at the wake-up time, recalculate its solution and schedule a new “wake up” event for the new acceptance time. The process may also wake itself up early for reevaluation by use of a timer (which can be viewed as just another process).

If the analog process identifies future analog events such as “crossings” or timer events (see Section 6.7.5) then it will schedule its wake-up event for the time of the first such event rather than the acceptance time. If the analog process is woken by such an analog event it will communicate any related events at that time and de-activate, rescheduling

1. Or derivatives w.r.t. time used to calculate the values.

it's wake-up for the next analog event or acceptance. Events to external processes generated from analog events are not communicated until the global simulation time reaches the time of the analog event.

If the time to acceptance is infinite then no wake-up event needs to be scheduled¹.

Analog processes are sensitive to changes in all variables and digital signals read by the process unless that access is only in statements 'guarded' by event expressions. For example the following code implements a simple digital to analog convertor:

```
module d2a(val,vo); // 16 bit D->A
    parameter    Vgain = 1.0/65536;
    input        val;
    wire [15:0] val;
    electrical   vo;
    analog begin
        V(vo) <+ Vgain * val;
    end
endmodule
```

The output voltage $V(vo)$ is reevaluated when any bit in *val* changes, which is not a problem if all the bits change simultaneously and no 'X' values occur. A practical design would require that the digital value is latched to avoid bad bit sequences, as in the following version:

```
module d2aC(clk,val,vo); // Clocked 16 bit D2A
    parameter    Vgain = 1.0/65536;
    input        clk;
    input        val;
    wire [15:0] val;
    electrical   vo;
    real         v_clkd;
    analog begin
        @(posedge clk) v_clkd = Vgain * val;
        V(vo) <+ v_clkd;
    end
endmodule
```

Since *val* is now guarded by the *@(posedge clock)* expression the analog block is not sensitive to changes in *val* and only reevaluates when *clk* changes.

1. The case when all derivatives are zero - the circuit is stable.

Macro processes can be evaluated separately but may be evaluated together¹, in which case, the wake up event for one process will cause the re-evaluation of all or some of the processes. Users should bear this in mind when writing mixed-signal code, as it will mean that the code should be able to handle re-evaluation at any time (not just at its own event times).

9.3.2.3 A/D boundary timing

In the analog kernel, time is a floating point value. In the digital kernel time is an integer value. Hence, A2D events generally do not occur exactly at digital integer clock ticks.

For the purpose of reporting results and scheduling delayed future events, the digital kernel converts analog event times to digital times such that the error is limited to half the precision base for the module where the conversion occurs. For the examples below the timescale is 1ns/1ns, so the maximum scheduling error when swapping a digital module for its analog counterpart will be 0.5ns.

Consequently an A2D event that results in a D2A event being scheduled with zero (0) delay, shall have its effect propagated back to the analog kernel with zero (0) delay.

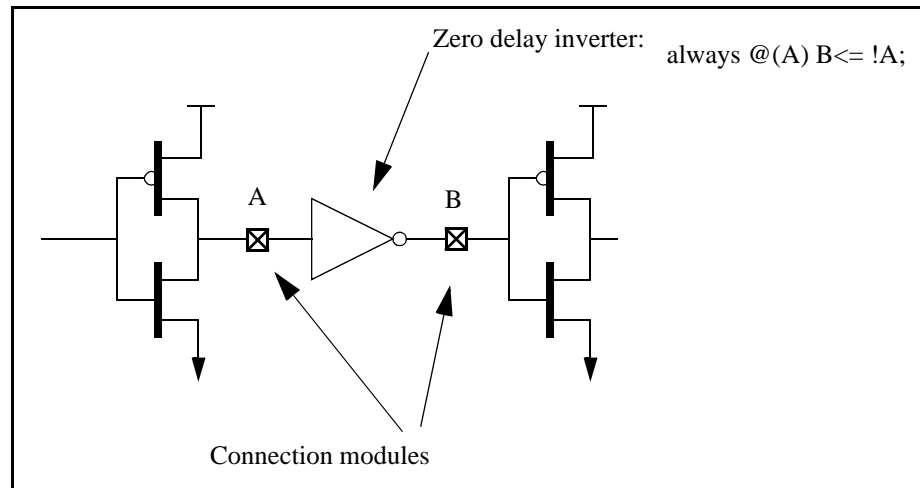


Figure 9-2 A zero delay inverter

If the circuit shown in Figure 9-2 is being simulated with a digital time resolution of $1e-9$ (one (1) nanosecond) then all digital events shall be reported by the digital kernel as having occurred at an integer multiple of $1e-9$. The A2D and D2A modules inserted are a simple level detector and a voltage ramp generator:

```
connectmodule a2d(i,o);
    parameter vdd = 1.0;
    logic      o;
    input      i;
```

1. This is implementation-dependent.


```

output      o;
reg         o;
electrical  i;
always begin @(cross(V(i) - vdd/2,+1)o = 1; end
always begin @(cross(V(i) - vdd/2,-1)o = 0; end
endmodule

connectmodule d2a(i, o);
    parameter vdd = 1.0;
    parameter slewrate = 2.0/1e9; // V/s
    input      i;
    output     o;
    electrical o;
    reg        qd_val, // queued value
           nw_val;
    real       et;      // delay to event
    real start_delay;   // .. to ramp start
    always @(driver_update i) begin
        nw_val = $driver_next_state(i,0); // assume one driver
        if (nw_val == qd_val) begin
            // no change (assume delay constant)
        end else begin
            et = $driver_delay(i,0) * 1e-9; // real delay
            qd_val = nw_val;
        end
    end
end
analog begin
    @(qd_val) start_delay = et - (vdd/2)/slewrate;
    V(o) <+ vdd * transition(qd_val,start_delay,vdd/slewrate);
end
endmodule

```

If connector A detects a positive threshold crossing, the resulting falling edge at connector B generated by the propagation of the signal through verilog inverter model shall be reported to the analog kernel with no further advance of analog time. The digital kernel will treat these events as if they occurred at the nearest nanosecond.

Example:

If A detects a positive crossing as a result of a transient solution at time 5.2×10^{-9} , the digital kernel shall report a rising edge at A at time 5.0×10^{-9} and falling edge at B at time 5.0×10^{-9} , but the analog kernel shall see the transition at B begin at time 5.2×10^{-9} , as shown in Figure 9-3. D2As fed with zero delay events cannot be preemptive, so the crossover on the return is delayed from the digital event; zero-delay inverters are not physically realizable devices.

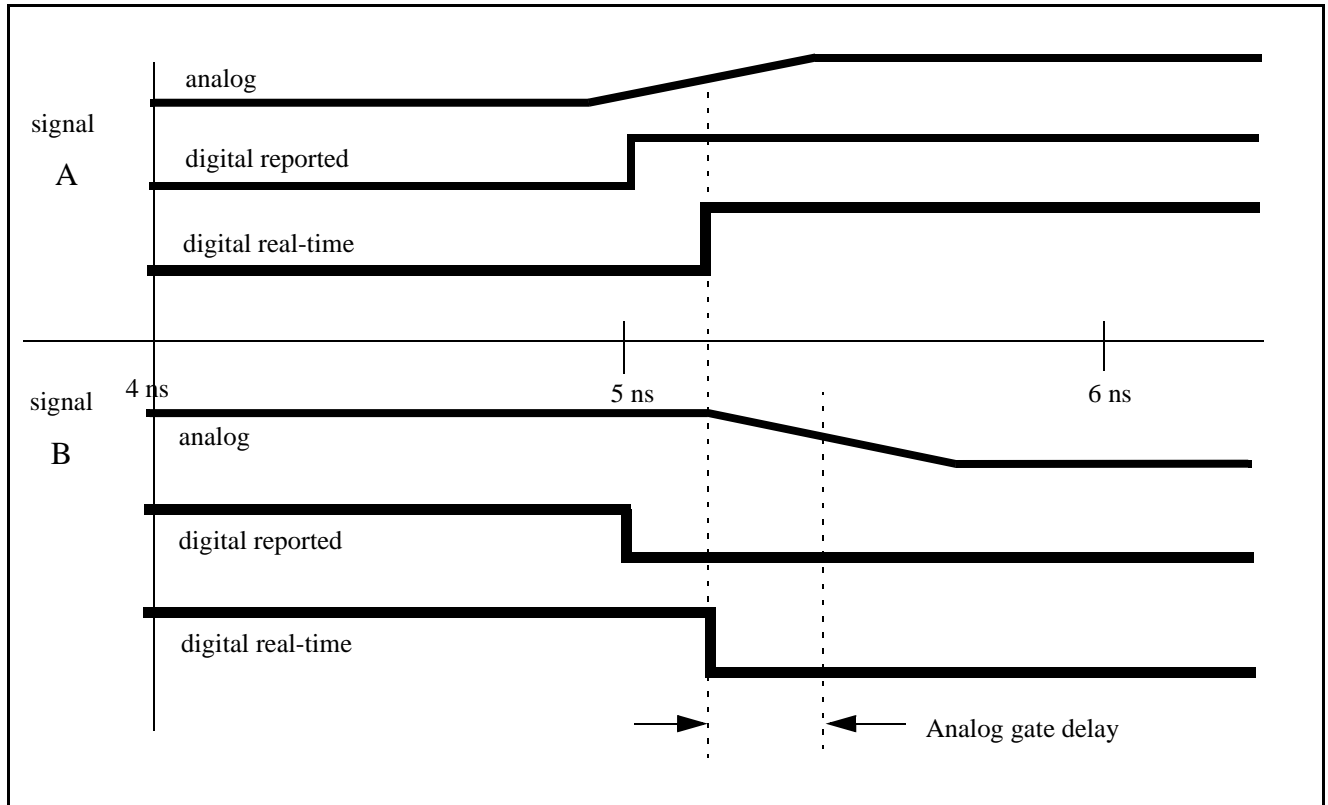


Figure 9-3 Zero delay transient solution times

If the inverter equation is changed to use a one unit delay (*always @(A) B <= #1 !A*), then the timing is as in Figure 9-4.

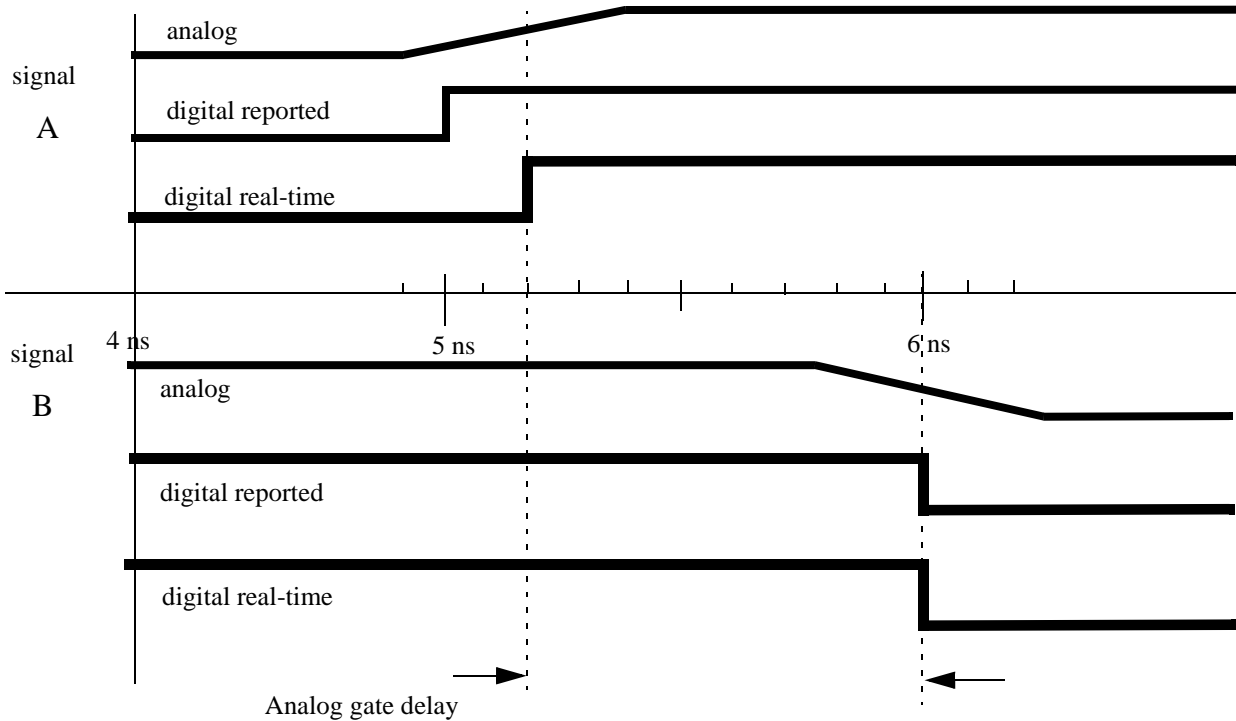


Figure 9-4 Unit delay transient solution times

9.3.3 The synchronization loop

Verilog-AMS uses a “conservative” simulation algorithm, the analog and digital processes that are managed by the simulation kernel are synchronized such that neither computes results that will invalidate signal values that have already been assigned; time never goes backwards. While the implementation of the simulator may have separate event queues for analog and digital events (see Section 9.3.4), it can be viewed as a single event queue logically with a common global time. Analog processes are similar to Verilog *initial* statements in that they start automatically at time zero. The event sequence for the transient simulation shown in Figure 9-4 would be as follows:

Time	Event Queue
4.9ns	Evaluate the first analog inverter
	Evaluate acceptance at 5.4ns, but schedule wake-up for 5.2 for crossing.
5.2ns	Evaluate crossing event
	The A2D logic sets the digital signal A, which triggers the evaluation of the non-blocking assign to B, which

	schedules the actual assignment for 6ns (rounded 1ns delay).
	D2A notices queued event and schedules wake-up for 5.75 via rampgen module.
	Schedule wake-up at 5.4ns (as previously calculated).
5.4ns	Evaluate acceptance Circuit evaluates stable, nothing scheduled.
5.75ns	D2A/rampgen process wake-up Start ramp in analog domain.
6.0ns	Non blocking assign performed (digital event). D2A may be sensitive, but doesn't need to do anything.
6.25ns	D2A/rampgen process wake-up Drive 0V to complete ramp. Nothing more to schedule.

Any events queued ahead of the current global event time may be cancelled. For instance, if the sequence above is interrupted by a change on the primary input before digital assignment takes place as shown in Figure 9-5.

Time	Event Queue
4.9ns	Evaluating the first analog inverter Evaluate acceptance at 5.4ns, but schedule wake-up for 5.2 for crossing.
5.2ns	Evaluate crossing event The A2D logic sets the digital signal A, which triggers the evaluation of the non-blocking assign to B, which schedules the actual assignment for 6ns (rounded 1ns delay). D2A notices queued event and changes value using transition filter. Schedule wake-up at 5.4ns (as previously calculated).
5.3ns	Analog event disturbs the solution Accept at 5.3ns. Cancel 5.4ns wake-up. New acceptance is 5.45ns, but schedule wake-up for crossing at 5.4ns.
5.4ns	Evaluate crossing event

The A2D logic sets the digital signal A, which triggers the evaluation of the non-blocking assign to B, which schedules the actual assignment for 6ns (rounded 1ns delay), cancelling previous event.

D2A detects the driver change and *qd_val* toggles back to 1 before the 0 propagates through the transition filter, so no analog change occurs at B.

Schedule wake-up at 5.45ns (as previously calculated).

5.45ns Evaluate acceptance

Circuit evaluates stable, nothing scheduled.

6.00ns Non blocking assign performed (digital event).

Value of B doesn't change.

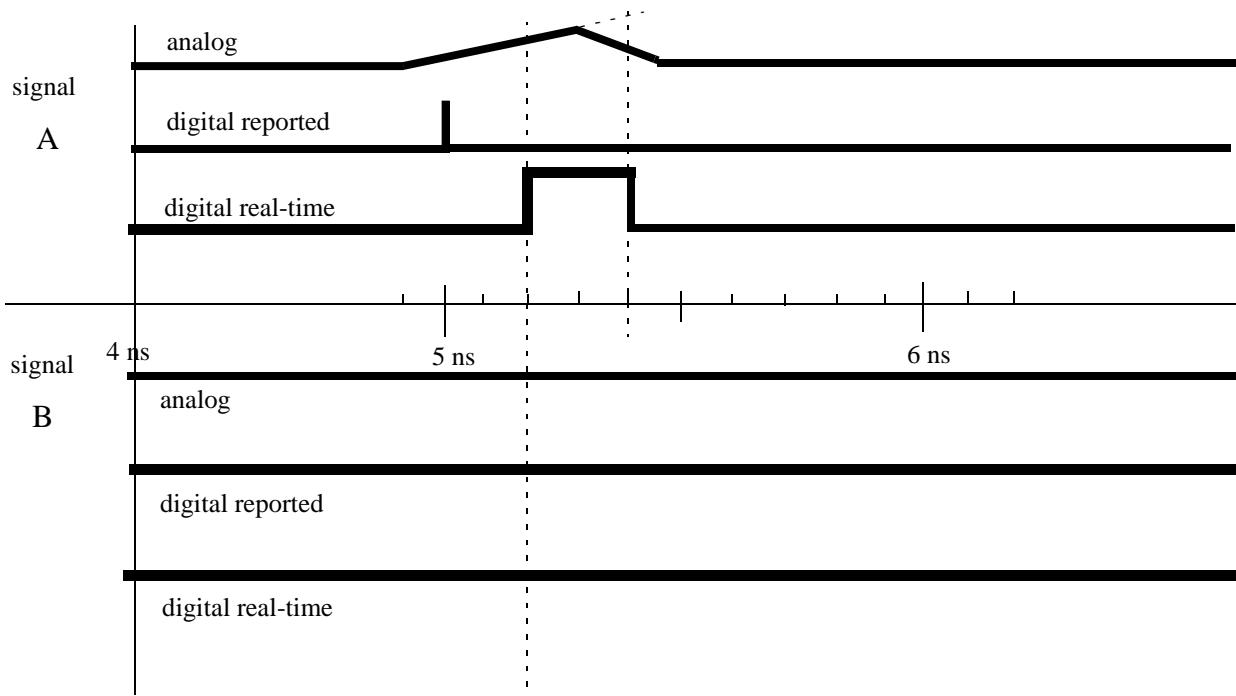


Figure 9-5 Transient solution times with glitch

If the cancelling event arrived after the ramp on B had started but before the assignment to the digital B, it is possible to see the glitch propagate back into the analog domain without an event appearing on B.

9.3.4 Synchronization and communication algorithm

Figure 9-6 is an abstract representation of how the analog engine simulating an analog macro process communicates and synchronizes with the digital engine and vice-versa.

The synchronization algorithm can exploit characteristics of the analog and digital kernels described in the next section. The arrows represent an engine moving from one synchronization point to another, which in the case of an analog macro-process involves one or more time-steps and in the case of a digital engine, involves once or more discrete times at which events are processed.

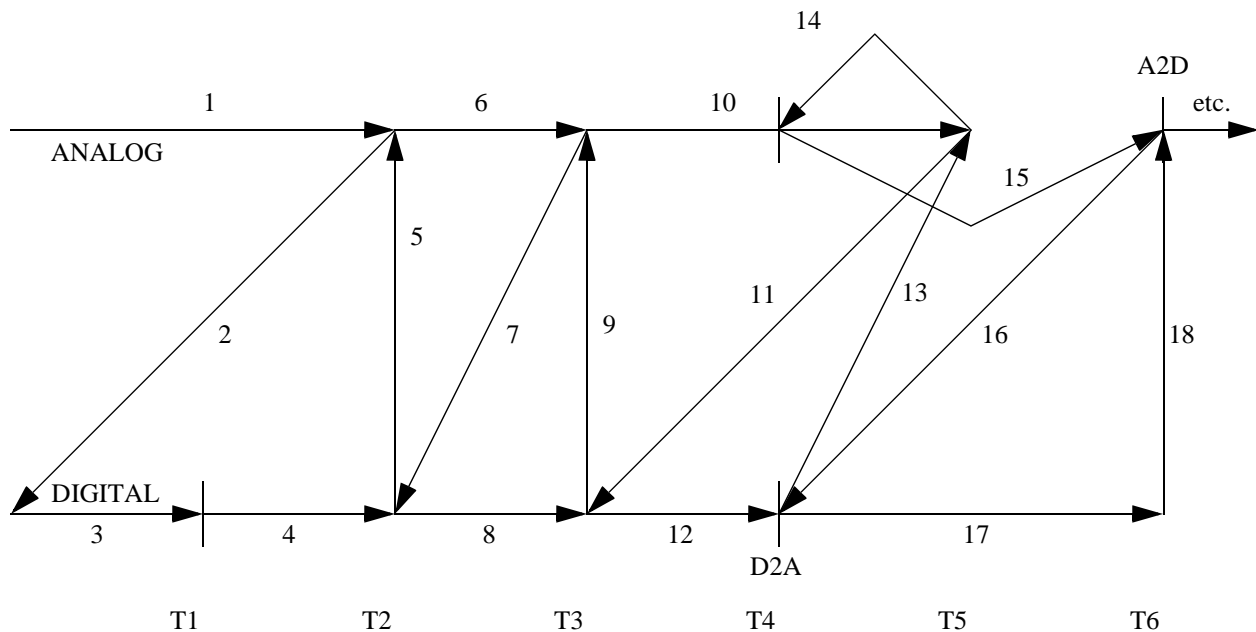


Figure 9-6 Sample run

1. The Analog engine begins transient analysis and sends state information (that it is good up to T2) to the Digital engine (1, 2).
2. The Digital engine begins to run using its own time steps (3); however, if there is no D2A event, the Analog engine is not notified and the digital engine continues to simulate until it can not advance its time without surpassing the time of the analog solution (4). Control of the simulation is then returned to the analog engine (5), which accepts at T2. This process is repeated (7, 8, 9, 10, and 11).
3. If the Digital engine produces a D2A event (12), control of the simulation is returned to the Analog engine (13). The analog engine accepts at the time of the D2A event (14, which may involve recalculating from T3). The Analog engine then calculates the next time step (15).

4. If the Analog engine produces an A2D event, it returns control to the Digital engine (16), which simulates up to the time of the A2D event, and then surrenders control (17 and 18).
5. This process continues until transient analysis is complete.

9.3.5 Assumptions about the analog and digital algorithms

1. Advance of time in a digital algorithm
 - A. The digital simulation has some minimum time granularity and all digital events occur at a time which is some integer multiple of that granularity.
 - B. The digital simulator can always accept events for a given simulation time provided it has not yet executed events for a later time. Once it executes events for a given time, it can not accept events for an earlier time.
 - C. The digital simulator can always report the time of the most recently executed event and the time of the next pending event.
2. Advance of time in an analog algorithm
 - A. The analog simulator advances time by calculating a sequence of solutions. Each solution has an associated time which, unlike the digital time, is not constrained to a particular minimum granularity.
 - B. The analog simulator can not tell for certain the time when the next solution converges. Thus, it can tell the time of the most recently calculated solution, but not the time of the next solution.
 - C. In general, the analog solution is a function of one or more previous solutions. Having calculated the solution for a given time, the analog simulator can either accept or reject that solution; it cannot calculate a solution for a future time until it has accepted the solution for the current time.
3. Analog to digital events
 - A. Analog to digital events are generated by conversion elements (which are analog/digital behavioral models) when evaluated by the analog simulator.
 - B. Analog events (e.g., `cross`, `initial_step`, and `final_step`) cause an analog solution of the time where they occur.
 - C. Thus, any analog to digital event is generated as the result of a particular transient solution. This means events can stay associated with the solution that produced them until they are passed to the digital simulator. Until then, they can be rejected along with the solution, if it is rejected.
4. Digital to analog events shall cause an analog solution of the time where they occur.

9.4 Scheduling semantics for the digital engine

The scheduling semantics for Verilog-HDL simulation are outlined in Section 5 of IEEE 1364-2001.

The digital engine of a Verilog-AMS mixed-signal simulator shall comply with that section except for the changes outlined in this section.

For mixed-signal simulation, the major change from Section 5 of IEEE 1364-2001 is that two new types of event must be supported by the event queue called the *explicit D2A* (digital-to-analog) *event*, and the *analog macro-process event*.

Explicit D2A events are created when a digital event occurs to which an analog block is *explicitly sensitive*. An analog block is explicitly sensitive to event expressions mentioned in an event control statement in that analog block.

Similarly, there is also the concept of the *implicit D2A event* that is created when a digital variable to which an analog block is *implicitly sensitive* changes value. An analog block is implicitly sensitive to all digital variable references that are not guarded by event control statements in that analog block.

An analog macro-process event is also created when either type of D2A event occurs. The analog macro-process event is associated with the analog macro-process that is sensitive to the D2A event. An analog macro-process event is evaluated by calling the analog engine to solve it. Note that implicit D2A events are not added to the stratified event queue, but as they directly cause an analog macro-process event, they effectively force a digital-analog synchronization when they occur.

9.4.1 The stratified event queue

The Verilog event queue is logically segmented into *seven* different regions. Events are added to any of the seven regions but are only removed from the *active* region. Regions 1b and 3b have been added for mixed-signal simulation.

1. Events that occur at the current simulation time and can be processed in any order. These are the *active* events.
- 1b. Explicit D2A events that occur at the current simulation time shall be processed after all the active events are processed.
2. Events that occur at the current simulation time, but that shall be processed after all the active and explicit D2A events are processed. These are the *inactive* events.
3. Events that have been evaluated during some previous simulation time, but that shall be assigned at this simulation time after all the active, explicit D2A and inactive events are processed. These are the non blocking assign update events.

- 3b. Analog macro-process events shall be processed after all active, explicit D2A events, inactive events and non blocking assign update events are processed.
4. Events that shall be processed after all the active, explicit D2A, inactive, non blocking assign update events and analog macro-process events are processed. These are the *monitor* events.
5. Events that occur at some future simulation time. These are the *future* events. Future events are divided into *future inactive events* and *future non blocking assignment update events*.

The processing of all the active events is called a *simulation cycle*.

The freedom to choose any active event for immediate processing is an essential source of nondeterminism in the Verilog HDL.

An *explicit zero delay* (#0) requires that the process be suspended and added as an inactive event for the current time so that the process is resumed in the next simulation cycle in the current time.

A non blocking assignment (see Section 9.2.2 of IEEE 1364-2001) creates a non blocking assign update event, scheduled for current or a later simulation time.

The **\$monitor** and **\$strobe** system tasks (see Section 17.1 of IEEE 1364-2001) create monitor events for their arguments. These events are continuously re-enabled in every successive time step. The monitor events are unique in that they cannot create any other events.

The call back procedures scheduled with PLI routines such as `tf_synchronize()` (see Section 25.58 of IEEE 1364-2001) or `vpi_register_cb(cb_readwrite)` (see Section 27.33 of IEEE 1364-1995) shall be treated as inactive events.

Note that A2D events must be analog event controlled statements (e.g., `@cross`, `@timer`). These are scheduled just like other event controlled statements in Verilog-HDL (e.g., `@posedge`).

9.4.2 The Verilog-AMS digital engine reference model

In all the examples that follow, T refers to the current simulation time of the digital engine, and all events are held in the event queue, ordered by simulation time.

```
while (there are events){
    if (no active events){
        if (there are inactive events){
            activate all inactive events;
        }
        else if (there are explicit D2A events) {
            activate all explicit D2A events;
        }
        }else if (there are non blocking assign update events){
```

```

        activate all non blocking assign update events;
    }else if (there are analog macro-process events) {
        activate all analog macro-process events;
    }else if (there are monitor events){
        activate all monitor events;
    }else {
        advance T to the next event time;
        activate all inactive events for time T;
    }
}
E =any active event;
if (E is an update event){
    update the modified object;
    add evaluation events for sensitive processes to event queue;
}else if (E is a D2A event) {
    evaluate the D2A
    modify the analog values
    add A2D events to event queue, if any
}else if (E is an analog macro-process event) {
    evaluate the analog macro-process
    modify the analog values
    add A2D events to event queue, if any
}else { /*shall be an evaluation event */
    evaluate the process;
    add update events to the event queue;
}
}

```

9.4.3 Scheduling implication of assignments

Assignments are translated into processes and events as follows.

9.4.3.1 Continuous assignment

A continuous assignment statement (Section 6 of IEEE 1364-2001) corresponds to a process, sensitive to the source elements in the expression. When the value of the

expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

9.4.3.2 Procedural continuous assignment

A procedural continuous assignment (which is the **assign** or **force** statement; see Section 9.3 of IEEE 1364-2001) corresponds to a process that is sensitive to the source elements in the expression. When the value of the expression changes, it causes an active update event to be added to the event queue, using current values to determine the target.

A **deassign** or a **release** statement deactivates any corresponding **assign** or **force** statement(s).

9.4.3.3 Blocking assignment

A blocking assignment statement (see Section 9.2.1 of IEEE 1364-2001) with a delay computes the right-hand side value using the current values, then causes the executing process to be suspended and scheduled as a future event. If the delay is 0, the process is scheduled as an inactive event for the current time.

When the process is returned (or if it returns immediately if no delay is specified), the process performs the assignment to the left-hand side and enables any events based upon the update of the left-hand side. The values at the time the process resumes are used to determine the target(s). Execution may then continue with the next sequential statement or with other active events.

9.4.3.4 Non blocking assignment

A non blocking assignment statement (see Section 9.2.2 of IEEE 1364-2001) always computes the updated value and schedules the update as a non blocking assign update event, either in this time step if the delay is zero or as a future event if the delay is nonzero. The values in effect when the update is placed on the event queue are used to compute both the right-hand value and the left-hand target.

9.4.3.5 Switch (transistor) processing

The event-driven simulation algorithm described in Section 5.4 of IEEE 1364-2001 depends on unidirectional signal flow and can process each event independently. The inputs are read, the result is computed, and the update is scheduled. The Verilog HDL provides switch-level modeling in addition to behavioral and gate-level modeling. Switches provide bi-directional signal flow and require coordinated processing of nodes connected by switches.

The Verilog HDL source elements that model switches are various forms of transistors, called **tran**, **tranif0**, **tranif1**, **rtran**, **rtranif0**, and **rtranif1**.

Switch processing shall consider all the devices in a bidirectional switch-connected net before it can determine the appropriate value for any node on the net, because the inputs and outputs interact. A simulator can do this using a relaxation technique. The simulator

can process tran at any time. It can process a subset of tran-connected events at a particular time, intermingled with the execution of other active events. Further refinement is required when some transistors have gate value x. A conceptually simple technique is to solve the network repeatedly with these transistors set to all possible combinations of fully conducting and nonconducting transistors. Any node that has a unique logic level in all cases has steady-state response equal to this level. All other nodes have steady-state response.

9.4.3.6 Processing explicit D2A events (region 1b)

An explicit D2A event is processed by evaluating the analog block that is sensitive to this event. This is so that the values used for the digital variables referenced inside the explicitly sensitive event control statement in the analog block are the values of those variables after region 1 has been processed, not the values of those variables just before region 3b is processed.

9.4.3.7 Processing analog macro-process events (region 3b)

An analog macro-process event is evaluated by calling the analog engine to solve the associated analog macro-process. Note that if multiple events for a particular analog macro-process are active, then a single evaluation of the analog macro-process shall consume all of these events from the queue.

The reason for processing analog macro-processes after regions 1-3 have been processed is to minimize the number of times analog macro-processes are evaluated, because such evaluations tend to be expensive.

Section 10

System tasks and functions

This section describes system tasks and functions available in Verilog-AMS HDL.

10.1 Environment parameter functions

The syntax for these functions are shown in Syntax 10-1.

```
environment_parameter_functions ::=
    $temperature
    | $abstime
    | $realtime [ ( real_number ) ]
    | $vt [ ( temperature_expression ) ]
```

Syntax 10-1—Syntax for the environment parameter functions

These functions return information about the current environment parameters as a real value.

\$temperature does not take any input arguments and returns the circuit's ambient temperature in Kelvin units.

\$abstime returns the absolute time, that is a real value number representing time in seconds.

\$realtime can have an optional argument which scales the time. If no argument is given, **\$realtime**'s return value is scaled to the ``time_unit` of the module which invoked it. If an argument is given, **\$realtime** shall divide the absolute time by the value of the argument (i.e., scale to the value specified in the argument). The argument for **\$realtime** follows the semantics of the ``time_unit`, that is it shall consist of an integer followed by a scale factor. Valid integers are: 1, 10, and 100; valid scale factors are: `s` (seconds), `ms` (milliseconds), `us` (microseconds), `ns` (nanoseconds), `ps` (picoseconds), and `fs` (femtoseconds).

\$vt can optionally have temperature (in Kelvin units) as an input argument and returns the thermal voltage (kT/q) at the given temperature. **\$vt** without the optional input temperature argument returns the thermal voltage using **\$temperature**.

Note: Previous Verilog-A modules using **\$realtime** may not produce the same results as in the past as **\$realtime** used to return time scaled to one (1) second. If the `time_unit` of the ``timescale` directive is

set to 1s, the behavior shall be the same. For analog blocks, the **\$abstime** function should typically be used, as it returns time in seconds.

10.2 \$random function

The syntax for this function is shown in Syntax 10-2.

```
random_function ::=  
    $random [ ( seed_expression ) ] ;
```

Syntax 10-2—Syntax for the random_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.

seed_expression controls the numbers **\$random** returns. *seed* shall be a **reg**, **integer**, or **time** variable. The *seed* value shall be assigned to this variable prior to calling **\$random**.

Examples:

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;
```

10.3 \$dist_ functions

The syntax for these functions are shown in Syntax 10-3.

```

distribution_functions ::=
    $digital_dist_functions ( args ) ;
| $rdist_uniform ( seed, start_expression, end_expression ) ;
| $rdist_normal ( seed, mean_expression, standard_deviation_expression ) ;
| $rdist_exponential ( seed, mean_expression ) ;
| $rdist_poisson ( seed, mean_expression ) ;
| $rdist_chi_square ( seed, degree_of_freedom_expression ) ;
| $rdist_t ( seed, degree_of_freedom_expression ) ;
| $rdist_erlang ( seed, k_stage_expression, mean_expression ) ;

seed ::=
    integer_variable_identifier

```

Syntax 10-3—Syntax for the probabilistic distribution functions

The following rules apply to these functions.

- All parameters to the system functions are real values, except for *seed* (which is defined by \$random()). For the \$rdist_exponential, \$rdist_poisson, \$rdist_chi_square, \$rdist_t, and \$rdist_erlang functions, the parameters *mean*, *degree_of_freedom*, and *k_stage* shall be greater than zero (0).
- Each of these functions returns a pseudo-random number whose characteristics are described by the function name, e.g., \$rdist_uniform returns random numbers uniformly distributed in the interval specified by its parameters.
- For each system function, the *seed* parameter is an inout parameter; that is, a value is passed to the function and a different value is returned. The system functions shall always return the same value given the same *seed*. This facilitates debugging by making the operation of the system repeatable. The argument for *seed* shall be an integer variable, which is initialized by the user and only updated by the system function. This ensures the desired distribution is achieved.
- All functions return a real value.
- In \$rdist_uniform, the *start* and *end* parameters are real inputs which bound the values returned. The *start* value shall be smaller than the *end* value.
- The *mean* parameter used by \$rdist_normal, \$rdist_exponential, \$rdist_poisson, and \$rdist_erlang is an real input which causes the average value returned by the function to approach the value specified.
- The *standard_deviation* parameter used by \$rdist_normal is a real input, which helps determine the shape of the density function. Using larger numbers for *standard_deviation* spreads the returned values over a wider range. Using a *mean* of zero (0) and a *standard_deviation* of one (1), \$rdist_normal generates Gaussian distribution.

- The *degree_of_freedom* parameter used by **\$rdist_chi_square** and **\$rdist_t** is a real input, which helps determine the shape of the density function. Using larger numbers for *degree_of_freedom* spreads the returned values over a wider range.

10.4 Simulation control system tasks

There are two simulation control system tasks, **\$finish** and **\$stop**.

10.4.1 \$finish

The syntax for this task is shown in Syntax 10-4.

```
finish_task ::=
    $finish [ ( n ) ] ;
```

Syntax 10-4—Syntax for the finish_task

\$finish simply makes the simulator exit. If an expression is supplied to this task, its value determines which diagnostic messages are printed before the prompt is issued, as shown in Table 10-1. One (1) is the default if no argument is supplied.

Table 10-1—Diagnostic messages

Parameter	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

10.4.2 \$stop

The syntax for this task is shown in Syntax 10-5.

```
stop_task ::=
    $stop [ ( n ) ] ;
```

Syntax 10-5—Syntax for the stop_task

\$stop causes simulation to be suspended at a converged timepoint. This task takes an optional expression argument (0, 1, or 2), which determines what type of diagnostic message is printed. The amount of diagnostic messages output increases with the value of *n*, as shown in Table 10-1.

10.5 File operation tasks

This section details the file operation tasks.

10.5.1 \$fopen

The syntax for this task is shown in Syntax 10-6.

```
file_open_task ::=
    integer multi_channel_descriptor = $fopen ( " file_name " ) ;
```

Syntax 10-6—Syntax for the file_open_task

\$fopen opens the file specified as an argument and returns a 32-bit multichannel descriptor which is uniquely associated with the file. It returns 0 if the file could not be opened for writing.

The multichannel descriptor can 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 which have been opened by **\$fopen**.

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.

10.5.2 \$fclose

The syntax for this task is shown in Syntax 10-7.

```
file_close_task ::=
    $fclose ( multi_channel_descriptor_identifer ) ;
```

Syntax 10-7—Syntax for the file_close_task

\$fclose closes the channels specified in the multichannel descriptor and does not allow any further output to the closed channels. **\$fopen** reuses channels which have been closed.

10.6 Display tasks

The syntax for these functions are shown in Syntax 10-8.

```
display_tasks ::=
    $strobe ( list_of_arguments ) ;
| $display ( list_of_arguments ) ;
| $monitor ( list_of_arguments ) ;
| $write ( list_of_arguments ) ;
```

Syntax 10-8—Syntax for the *display_tasks*

The following rules apply to these functions.

- **\$strobe** provides the ability to display simulation data when the simulator has converged on a solution for all nodes.
- **\$strobe** displays its arguments in the same order they appear in the argument list. Each argument can be a quoted string, an expression which 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 the character to follow is a literal or non-printable character (see Table 10-2).
 - The special character `%` indicates the next character shall be interpreted as a format specification which establishes the display format for a subsequent expression argument (see Table 10-3). For each `%` character which appears in a string, a corresponding expression argument shall be supplied after the string.
 - The special character string `%%` indicates the display of the percent sign character (`%`) (see Table 10-2).
- Any null argument produces a single space character in the display. (A *null argument* is characterized by two adjacent commas `(,)` in the argument list.)
- When **\$strobe** is invoked without arguments, it simply prints a newline character.

The **\$display** task provides the same capabilities as **\$strobe**. The **\$write** task provides the same capabilities as **\$strobe**, but with no newline. The **\$monitor** task provides the same capabilities as **\$strobe**, but outputs only when a parameter changes.

10.6.1 Escape sequences for special characters

The escape sequences shown in Table 10-2, when included in a string argument, print special characters.

Table 10-2— Escape sequences for printing special characters

<code>\n</code>	The newline character
<code>\t</code>	The tab character
<code>\\</code>	The <code>\</code> character
<code>\"</code>	The <code>"</code> character
<code>\ddd</code>	A character specified by 1 to 3 octal digits
<code>%%</code>	The <code>%</code> character

10.6.2 Format specifications

Table 10-3 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` and `%%`) which appears in a string, a corresponding expression shall follow the string in the argument list, except a null argument. The value of the expression replaces the format specification when the string is displayed.

Table 10-3— Escape sequences for format specifications

<code>%h</code> or <code>%H</code>	Display in hexadecimal format
<code>%d</code> or <code>%D</code>	Display in decimal format
<code>%o</code> or <code>%O</code>	Display in octal format
<code>%b</code> or <code>%B</code>	Display in binary format
<code>%c</code> or <code>%C</code>	Display in ASCII character format
<code>%m</code> or <code>%M</code>	Display hierarchical name
<code>%s</code> or <code>%S</code>	Display as a string

Any expression argument which has no corresponding format specification is displayed using the default decimal format in **\$strobe**.

The format specifications in Table 10-4 are used for real numbers and have the full formatting capabilities available in the C language. For example, the format specification `%10.3g` sets a minimum field width of 10 with three (3) fractional digits.

Table 10-4— Format specifications for real numbers

<code>%e</code> or <code>%E</code>	Display 'real' in an exponential format
<code>%f</code> or <code>%F</code>	Display 'real' in a decimal format
<code>%g</code> or <code>%G</code>	Display 'real' in exponential or decimal format, whichever format results in the shorter printed output

10.6.3 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 which invokes the system task containing the format specifier. This is useful when there are many instances of the module which call the system task. One obvious application is timing check messages in a flip-flop or latch module; the `%m` format specifier pinpoints the module instance responsible for generating the timing check message.

10.6.4 String format

The `%s` format specifier is used to print ASCII codes as characters. For each `%s` specification which appears in a string, a corresponding parameter shall 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 shall be right-justified so 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 (0) are never printed.

10.7 Announcing discontinuity

The `$discontinuity` function is used to give hints to the simulator about the behavior of the module so the simulator can control its simulation algorithms to get accurate results in exceptional situations. This function does not directly specify the behavior of the module. `$discontinuity` shall be executed whenever the analog behavior changes discontinuously.

The general form is

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

where *constant_expression* indicates the degree of the discontinuity.

`$discontinuity(i)` implies 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.

Note: Because discontinuous behavior can cause convergence problems, discontinuity shall be avoided whenever possible.

The filter functions (`transition()`, `slew()`, `laplace()`, etc.) can be used to smooth discontinuous behavior. However, in some cases it is not possible to implement the desired functionality using these filters. In those cases, the `$discontinuity` function shall be executed when the signal behavior changes abruptly.

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

*Examples:**Example 1*

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 ;
    if (V(pin,nin) >= 0)
      I(c1,c2) <+ V(c1,c2)/r;
    else
      I(c1,c2) <+ 0 ;
    end
  endmodule

```

In this example, `cross()` controls the time step so the time when the relay changes position is accurately resolved. It also triggers the `$discontinuity` function, which 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.

Example 2

Another example is a source which 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 to eliminate problems resulting 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 = $abstime ;
      $discontinuity;
    end

    @(timer(period/2, period)) begin
      slope = -1 ;
      offset = $abstime;
      $discontinuity ;
    end
  end

```

```

        V(out) <+ amplitude*slope*
            (4*($abstime - offset)/period - 1);
    end
endmodule

```

10.8 Time related functions

The `$bound_step()` function puts a bound on the next time step. It does not specify exactly what the next time step is, 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 general form is

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

where *expression* is a required argument and represents the maximum timestep the simulator can advance.

Examples:

The example below implements a sinusoidal voltage source and uses the `$bound_step()` function to assure 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*$abstime) + offset;
        $bound_step(0.05/freq);
    end
endmodule

```

For details on the `last_crossing()` function, see Section 4.4.10.

Section 11

Compiler directives

All Verilog-AMS 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 section describes the following compiler directives:

```
`default_discipline
`default_transition
`define
`else
`endif
`ifdef
`include
`resetall
`undef
```

Those compiler directives defined in *IEEE 1364-1995 Verilog HDL* are also supported.

11.1 ``default_discipline`

The scope of this directive is similar to the scope of the ``define` compiler directive, although it can be used only outside of module definitions. The default discipline is applied to all discrete signals without a discipline declaration that appear in the text stream following the use of the ``default_discipline` directive, until either the end of the text stream or another ``default_discipline` directive with the qualifier (if applicable) is found in the subsequent text, even across source file boundaries. Therefore, more than one ``default_discipline` directive can be in force simultaneously, provided each differs in qualifier.

In addition to ``reset_all`, if this directive is used without a discipline name, it turns off all currently active default disciplines without setting a new default discipline. Subsequent discrete signals without a discipline shall be associated with the empty discipline. Syntax 11-1 shows the syntax for this directive.


```

default_discipline_directive ::=
    `default_discipline [discipline_identifier [ qualifier ] ]

qualifier ::=
    integer | real | reg | wreal
    wire | tri | wand | triand | wor | trior | triereg |
    tri0 | tri1 | supply0 | supply1

```

Syntax 11-1—Syntax for the default discipline compiler directive

Example:

```

`default_discipline logic

module behavnnand(in1, in2, out);
    input in1, in2;
    output out;
    reg out;

    always begin
        out = ~(in1 && in2);
    end

endmodule

```

This example illustrates the usage of the ``default_discipline` directive. The nets `in1`, `in2`, and `out` all have discipline `logic` by default.

There is a precedence of compiler directives; the more specific directives have higher precedence over general directives.

11.2 ``default_transition`

The scope of this directive is similar to the scope of the ``define` compiler directive although it can be used only outside of module definitions. This directive specifies the default value for rise and fall time for the transition filter (see Section 4.4.8). There are no scope restrictions for this directive. The syntax for this directive is shown in Syntax 11-2.

```

default_transition_compiler_directive ::=
    `default_transition transition_time

transition_time ::=
    constant_expression

```

Syntax 11-2—Syntax for default transition compiler directive

transition_time is a real value.

For all transition filters which follow this directive and do not have rise time and fall time arguments specified, *transition_time* is used as the default rise and fall time values. If another **`default_transition** directive is encountered in the subsequent source description, the transition filters following the newly encountered directive derive their default rise and fall times from the transition time value of the newly encountered directive. In other words, the default rise and fall times for a transition filter are derived from the *transition_time* value of the directive which immediately precedes the transition filter.

If a **`default_transition** directive is not used in the description, *transition_time* is controlled by the simulator.

11.3 **`define and `undef**

A text macro substitution facility allows meaningful names to 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, as only one place in the source description needs to be altered if the value of the constant changed.

11.3.1 **`define**

`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 *`text_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 shown in Syntax 11-3.

```
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 }
```

Syntax 11-3—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 shall be preceded by a backslash (`\`). The first newline not preceded by a backslash shall 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 (i.e., a comment specified with the characters `//`) is included in the text, 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 shown in Syntax 11-4.

```
text_macro_usage ::=
    `text_macro_identifier [ ( list_of_actual_arguments ) ]

list_of_actual_arguments ::=
    actual_argument { , actual_argument }

actual_argument ::=
    expression
```

Syntax 11-4—Syntax for text macro usage

For a macro without an argument, the text is substituted “as is” for every occurrence of ``text_macro`. However, a text macro with one or more arguments shall 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; i.e., there are no scope restrictions.

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)
// Given the above macro the following uses
`var_adc(2) g121 (q21, n10, n11);
`var_adc(5) g122 (q22, n10, n11);

// shall result in:
adc #(2) g121 (q21, n10, n11);
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 1: Text macro names can not be the same as compiler directive keywords.

Note 2: Text macro names can re-use names being used as ordinary identifiers. For example, `signal_name` and ``signal_name` are different.

Note 3: 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.

11.3.2 ``undef`

``undef` undefines a previously defined text macro. An attempt to undefine a text macro which was not previously defined by a ``define` compiler directive can result in a warning. The syntax for ``undef` is shown in Syntax 11-5.

<pre> undefine_compiler_directive ::= `undef text_macro_identifier </pre>

Syntax 11-5—Syntax for undef compiler directive

An undefined text macro has no value.

11.4 ``ifdef, `else, `endif`

These conditional compilation compiler directives are used to optionally include lines of a Verilog-AMS HDL source description during compilation. ``ifdef` checks for the definition of a variable name. If the variable name is defined, 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 can appear anywhere in the source description.

``ifdef`, ``else`, and ``endif` can be useful when:

- selecting different representations of a module such as behavioral, structural, or mixed level;
- choosing different timing or structural information; or
- selecting different stimulus for a given simulation run.

Syntax 11-6 shows the syntax for ``ifdef`, ``else`, and ``endif`.

```
conditional_compilation_directive ::=  
    `ifdef text_macro_name  
        first_group_of_lines  
    [ `else  
        second_group_of_lines  
    `endif ]
```

Syntax 11-6—Syntax for conditional compilation directives

text_macro_name is a Verilog-AMS HDL identifier. The *first_group_of_lines* and the *second_group_of_lines* are parts of a Verilog-AMS HDL source description. The ``else` compiler directive and the *second_group_of_lines* are optional.

``ifdef`, ``else`, and ``endif` work in the following manner:

- When an ``ifdef` is encountered, *text_macro_name* is tested to see if it is defined as a *text_macro_name* by ``define` within the Verilog-AMS HDL source description.
- If *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 *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 1: Any group of lines the compiler ignores still needs to follow the Verilog-AMS HDL lexical conventions for white space, comments, numbers, strings, identifiers, keywords, and operators.

Note 2: These compiler directives can be nested.

11.5 ``include`

This 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. ``include` can be used to include global or

commonly used definitions and tasks without encapsulating repeated code within module boundaries.

This directive can be useful in the following situations:

- providing an integral part of configuration management;
- improving the organization of Verilog-AMS HDL source descriptions; or
- facilitating the maintenance of Verilog-AMS HDL source descriptions.

The syntax for ``include` is shown in Syntax 11-7.

```
include_compiler_directive ::=  
    `include "filename"
```

Syntax 11-7—Syntax for include compiler directive

The compiler directive ``include` can be specified anywhere within the Verilog-AMS 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 can appear on the same line as the ``include` compiler directive.

A file included in the source using ``include` can contain other ``include` compiler directives. The number of nesting levels for included files are finite.

Examples:

```
`include "parts/count.v"  
`include "fileA"  
`include "fileB" // including fileB
```

Note: Implementations can limit the maximum number of levels to which include files can be nested, but this limit shall be a minimum of 15 levels.

11.6 ``resetall`

When the ``resetall` compiler directive is encountered during compilation, all compiler directives are set to their default values. This is useful for ensuring only those directives which are desired in compiling a particular source file are active.

To do so, place ``resetall` at the beginning of each source text file, followed immediately by the directives desired in the file.

11.7 Predefined macros

Verilog-AMS HDL supports a predefined macro to allow modules to be written that work with both *IEEE 1364-1995 Verilog HDL* and Verilog-AMS HDL. The predefined macro is called `__VAMS_ENABLE__`.

This macro shall always be defined during the parsing of Verilog-AMS source text. Its purpose is to support the creation of modules which are both legal Verilog and Verilog-AMS. The Verilog-AMS features of such modules are made visible only when the `__VAMS_ENABLE__` macro has previously been defined.

Example:

```
module not_gate(in, out);
  input in;
  output out;
  reg out;
  `ifdef __VAMS_ENABLE__
    parameter integer del = 1 from [1:100];
  `else
    parameter del = 1;
  `endif
    always @ in
      out = #del !in;
endmodule
```

Annex B

Keywords

This annex contains the list of all keywords used in Verilog-AMS HDL.

B.1 All keywords

Keywords are predefined nonescaped identifiers which define Verilog-AMS HDL language constructs. An escaped identifier shall not be treated as a keyword.

abs	endfunction	ln	slew
absdelay	endnature	log	small
acos	endprimitive	macromodule	specify
acosh	endspecify	max	specparam
ac_stim	endtable	medium	sqrt
always	endtask	min	strong0
analog	event	module	strong1
analysis	exclude	nand	supply0
and	exp	nature	supply1
asin	final_step	negedge	table
asinh	flicker_noise	net_resolution	tan
assign	floor	nmos	tanh
atan	flow	noise_table	task
atan2	for	nor	time
atanh	force	not	timer
begin	forever	notif0	tran
branch	fork	notif1	tranif0
buf	from	or	tranif1
bufif0	function	output	transition
bufif1	generate	parameter	tri
case	genvar	pmos	tri0
casex	ground	posedge	tri1
casez	highz0	potential	triand
ceil	highz1	pow	trior
cmos	hypot	primitive	trireg
connectrules	idt	pull0	vectored
cos	idtmod	pull1	wait
cosh	if	pullup	wand
cross	ifnone	pulldown	weak0
ddt	inf	rcmos	weak1
deassign	initial	real	while
default	initial_step	realtime	white_noise
defparam	inout	reg	wire
disable	input	release	wor
discipline	integer	repeat	wreal
driver_update	join	rnmos	xnor
edge	laplace_nd	rpmos	xor
else	laplace_np	rtran	zi_nd
end	laplace_zd	rtranif0	zi_np
enddiscipline	laplace_zp	rtranif1	zi_zd
endcase	large	scalared	zi_zp
endconnectrules	last_crossing	sin	
endmodule	limexp	sinh	

B.2 Discipline/nature

Discipline and nature keywords are predefined nonescaped identifiers which define and are only applicable to Verilog-AMS HDL discipline and nature constructs. All discipline and nature keywords are used between the Verilog-AMS HDL keywords **discipline** and **enddiscipline** and between the keywords **nature** and **endnature**. An escaped identifier shall not be treated as a discipline or nature keyword.

abstol
access
continuous
ddt_nature
discrete
domain
idt_nature
units

B.3 Connect rules

Connect rules keywords are predefined nonescaped identifiers which define and are only applicable to Verilog-AMS HDL connect constructs. All connect rules keywords are used between the Verilog-AMS HDL keywords **connectrules** and **endconnectrules**. An escaped identifier shall not be treated as a connect_rules keyword.

connect
merged
resolveto
split

Annex C

Analog language subset

Prior to the release of Verilog-AMS HDL, the OVI board approved an analog-only specification called *Verilog-A v1.0*. With the release of Verilog-AMS HDL, the “official” *Verilog-A LRM* is no longer supported as it is included as part of the Verilog-AMS HDL specification. This annex defines a working subset of Verilog-AMS HDL for analog-only products.

C.1 Verilog-AMS introduction

This section previews Verilog-A and its language features.

C.1.1 Verilog-A overview

This Verilog-A subset defines a behavioral language for analog only systems. Verilog-A is derived from the *IEEE 1364-1995 Verilog HDL* specification using a minimum number of constructs for analog and mixed-signal behavioral descriptions. This Annex is intended to cover the definition and semantics of Verilog-A as proposed by OVI.

The intent of Verilog-A is to let designers of analog systems and integrated circuits create and use modules which encapsulate high-level behavioral 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. These behavioral descriptions can be used in many disciplines such as electrical, mechanical, fluid dynamics, and thermodynamics.

Verilog-A has been 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 terminals. 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.

C.1.2 Verilog-A language features

The Verilog-A subset provides access to a salient set of features of the full modeling language that allow analog designers the ability to model analog systems:

- Verilog-A modules are compatible with Verilog-AMS HDL.
- Analog behavioral modeling descriptions are contained in a separate analog block.
- Branches can be named for easy selection and access.
- Parameters can be specified with valid range limits.
- Systems can be modeled by using expressions consisting of operators, variables, and signals:
 - 1.a full set of operators including trigonometric functions, integrals, and derivatives;
 - 2.a set of waveform filters to modify the waveform results for faster and more accurate simulation like transition, slew, Laplace, and Z-domain;
 - 3.a set of events to control when certain code is simulated;
 - 4.selection of the simulation time step for simulation control;
 - 5.support for accessing SPICE primitives from within the language.

C.2 Lexical conventions

With the exception of certain keywords required for Verilog-AMS HDL, Section 2 is applicable to both Verilog-A and Verilog-AMS HDL. All Verilog-AMS HDL keywords shall be supported by Verilog-A as reserved words, but *IEEE 1364-2001 Verilog HDL* and Verilog-AMS HDL specific keywords are not used in Verilog-A. The following Verilog-AMS HDL keywords are not required to be supported for a fully compliant Verilog-A subset:

- From Section 2.5, Numbers: support for x and z values is limited in the analog block only to mixed signal, as defined in Section 8.3.2.
- From Section 2.6, Strings: support for regs and strings is limited to the digital context only.
- From Section 2.7.2, Keywords: certain keywords are not applicable in Verilog-A, as defined in Annex C.15.

C.3 Data types

The data types of Section 3 are applicable to both Verilog-AMS HDL and Verilog-A with the following exceptions:

- From Section 3.4.2.2, Domain binding: the domain binding type `discrete` shall be an error in Verilog-A.
- From Section 3.5, Real net declarations: the `wreal` data type is not supported in Verilog-A.
- From Section 3.6, Default discipline: the ``default_discipline` compiler directive is not supported in Verilog-A. All Verilog-A modules shall have a discipline defined for each module.

Note: This feature allows the use of digital modules in Verilog-AMS HDL without editing them to add a discipline.

C.4 Expressions

The expressions defined in Section 4 are applicable to both Verilog-AMS HDL and Verilog-A with the following exception:

The case equality operators (`===`, `!==`) are not supported in Verilog-A.

C.5 Signals

The signals defined in Section 5 are applicable to both Verilog-AMS HDL and Verilog-A.

C.6 Analog behavior

The analog behavior defined in Section 6 are applicable to both Verilog-AMS HDL and Verilog-A with the following exceptions:

- No digital behavior or events are supported in Verilog-A.
- `casex` and `casez` are not supported in Verilog-A.

C.7 Hierarchical structures

The hierarchical structure defined in Section 7 is applicable to both Verilog-AMS HDL and Verilog-A, except support for *real value ports* is only applicable to Verilog-AMS HDL and *IEEE 1364-1995 Verilog HDL* (see Section 7.3.3).

C.8 Mixed signal

This section only applies to Verilog-AMS HDL.

C.9 Scheduling semantics

The analog simulation cycle is applicable to both Verilog-AMS HDL and Verilog-A. The mixed-signal simulation cycle from Section 9.2 is only applicable to Verilog-AMS HDL.

C.10 System tasks and functions

The system tasks and functions in Section 10 are applicable to both Verilog-AMS HDL and Verilog-A.

C.11 Compiler directives

The compiler directives of Section 11 are applicable to both Verilog-AMS HDL and Verilog-A.

Verilog-A also supports the ``default_function_type_analog` directive, which allows user-defined functions to be treated as analog functions in Verilog-A if they do not have the key word **analog** as part of the definition. This is provided for backwards compatibility.

C.12 Using VPI routines

The analog behavior defined in Section 12 are applicable to both Verilog-AMS HDL and Verilog-A.

C.13 VPI routine definitions

The analog behavior defined in Section 13 are applicable to both Verilog-AMS HDL and Verilog-A.

C.14 Syntax

This annex (Annex A) defines the differences between Verilog-AMS HDL and Verilog-A. Annex A defines the BNF for Verilog-AMS HDL.

C.15 Keywords

The keywords in this annex (Annex B) are the complete set of Verilog-AMS HDL keywords, including those from *IEEE 1364-2001 Verilog HDL*. The following keywords as defined in this LRM are not used by Verilog-A:

- From Annex B.1, All keywords:

```
connectmodule
connectrules
driver_update
endconnectrules
net_resolution
wreal
```

- From Annex B.2, Discipline/nature:

All keywords in this section are supported in Verilog-A.

- From Annex B.3, Connect rules:

```
connect
merged
split
resolvedto
```

Note: All keywords of Verilog-AMS HDL are reserved words for Verilog-A.

C.16 Standard definitions

The definitions of Annex D are applicable to both Verilog-AMS HDL and Verilog-A, with the exception of those disciplines with a domain of **discrete**.

C.17 SPICE compatibility

Annex E defines the SPICE compatibility for both Verilog-A and Verilog-AMS HDL.

C.18 Changes from previous Verilog-A LRM versions

As part of the Verilog-AMS HDL development, some changes and clarifications have occurred to the current Verilog-A subset. Most of the changes resulted in clarifications or additional capability; but some new compatibility issues now exist. This subsection highlights some of the key differences. The syntax and semantics of this document supersede any syntax, semantics, or interpretations of the original document. Table C.1 lists the changes from Verilog-A LRM v1.0 to v2.0. Table C.2 lists the changes from Verilog-A LRM v2.0 to v2.1.

Table C.1—Changes from v1.0 to v2.0 syntax

Feature	OVI Verilog-A v1.0	OVI Verilog-AMS v2.0	Change type
Analog time	\$realtime	\$abstime	new
Ceiling operator	N/A	ceil (expr)	new
Floor operator	N/A	floor (expr)	new
Circular integrator	N/A	idtmod (expr)	new
Expression looping	N/A	genvar	new
Distribution functions	\$dist_functions() Integer based functions	\$rdist_functions() Real value equivalents to \$dist_functions()	new
Empty discipline	predefined as type wire	type not defined	default definition
Implicit nodes	'default_nodetype discipline_identifier default: wire	default type: empty discipline, no domain type	default definition
initial_step	default = TRAN	default = ALL	default definition
final_step	default = TRAN	default = ALL	default definition
Analog ground	no definition	now a declaration statement	definition
\$realtime	\$realtime :timescale =1 sec	\$realtime :timescale= 'timescale def=1n, see \$abstime	definition
Array setting	aa[0:1] = {2.1 = (1), 4.5 = (2)}	aa[0:1] = {2.1,4.5}	syntax
Discontinuity function	discontinuity (x)	\$discontinuity (x)	syntax
Limiting exponential function	\$limexp(expression)	limexp (expression)	syntax
Port branch access	I(a,a)	I(<a>)	syntax
Timestep control (maximum stepsize)	bound_step (const_expression n)	\$bound_step(expr)	syntax
Continuous waveform delay	delay ()	absdelay ()	syntax

Feature	OVI Verilog-A v1.0	OVI Verilog-AMS v2.0	Change type
User-defined analog functions	function	analog function See Section C.11	syntax
Discipline domain	N/A, assumed continuous	now continuous(default) and discrete	Extension
k scalar (10^3)	N/A, only “K” supported	now supported	Extension
Module keyword	module	module or macromodule	Extension
Modulus operator	integers only	now supports integer and reals	Extension
Time tolerance on timer functions	N/A	supports additional time tolerance argument for timer()	Extension
Time tolerance on transition filter	N/A	supports additional time tolerance argument for transition()	Extension
<code>`default_nodetype</code>	<code>`default_nodetype</code>	<code>`default_discipline</code>	Obsolete
Forever statement	forever	N/A	Obsolete
Generate statement	generate	N/A	Obsolete
Null statement	;	Limited to case, conditional, and event statements (see syntax)	Obsolete

Table C.2— Changes from v2.0 to v2.1

Item	Description/Issue	Section
1	Clarification on when range checking for parameters is done. Range check will be done only on the final value of the parameter for that instance.	Section 3.2.2
2	Not to use “max” and use “maxval” instead since max is a keyword	Section 3.4.1.1, Section 3.4.2.6
3	Support of user-defined attributes to disciplines similar to natures has been added. This would be a useful way to pass information to other tools reading the Verilog-AMS netlist	Section 3.4.2, Section 3.4.2.7
4	LRM specifies TRI and WIRE as aliases. The existing AMS LRM forces nets with wiretypes other than wire or tri to become digital, but in many cases these are really interconnect also. If they are tied to behavioral code they will become digital but if they are interconnected, we should not force them until after discipline resolution. This is needed if you have configs where the blocks connected to the net can change between analog and digital. If we force these nets to be digital we force unneeded CMs when blocks are switched to analog.	Section 3.4.2.4, Section 3.5
5	Setting an initial value on net as part of the net declaration.	Section 3.4.3 Syntax 3-6, Section 3.4.3.1
6	Initial value of wreal to be set to 0.0 if the value has not been determined at $t = 0$.	Section 3.5

Item	Description/Issue	Section
7	Clarification on the usage of `default_discipline` and default discipline for analog and digital primitives. Analog primitives will have default discipline as electrical, whereas digital primitives shall use the `default_discipline` declaration. `default_discipline` explanation moved to the section along with other compiler directives and clarification of impact on `reset_all` on this. The usage of word 'scope' is clarified to be used as the scope of the application of the compiler directive, and not as a scope argument.	Section 3.6, Section 3.6.1, Section 3.7 Section 11.1, Section 11.2
8	Reference to derived disciplines to be removed as current BNF does not support the syntax	Section 3.8
9	Reworked discipline and nature compatibility rules for better clarity.	Section 3.8
10	Removed the reference to neutral discipline since wire can be used in the same context.	Section 3.8
11	absdelay instead of delay	Section 4.4.14
12	Array declaration wrongly specified before the variable identifier. For variables, array specification is written after the name of the variable.	Section 6.5.2 (Example)
13	@(final_step) without arguments should not have parenthesis	Section 6.7.4, Table 6-1
15	@(final_step) for DCOP should be 1	Section 6.7.4, Table 6-1
16	Examples to be fixed to use assign for wreal and use wreal in instantiation, and also add a top level block for example in 7.3.3, and the testbench use wreal .	Section 7.3.3, Section 3.5
17	Clarification on the port bound semantics in explaining the hierarchical structure for a port with respect to vpiLoConn and vpiHiConn and clarification on driver and receiver segregation	Section 8.2.3
18	Figure should have NetC.c_out instead of NetC.b_out	Section 8.2.3
19	Mixed-signal module examples to use case syntax with X & Z instead of "==" for value comparison	Section 8.3.2
20	Clarification on accessing discrete nets and variables and X & Z bits in the analog context.	Section 8.3.2
21	Adding Support for 'NaN & X' into Verilog-AMS. Contribution of these values to a branch would be an error; however, analog variables should be able to propagate this value. Added a section regarding NaN	Section 8.3.2, Section 8.3.2.1
22	The diagram corresponding to the bidir model has been reworked, and the example module shown for bidir will match the corresponding figure.	Section 8.6
23	Rework on <i>connect-resolveto</i> syntax section to clarify the rules	Section 8.7.2.1
24	Use merged instead of merge	Section 8.8.1
25	Support for digital primitive instantiation in an analog block. Port names are created for the ports of the digital primitives, and these digital port names cannot be used in child instantiations.	Section 8.8.5.1

Item	Description/Issue	Section
26	Net resolution function has been removed and replaced with ‘Receiver Net Resolution’. Reintroduced the assign statement syntax.	Section 8.10.5
27	Corrections to the connect module example using the driver access function. The errors in the example have been corrected to make it syntactically and semantically correct	Section 8.10.6
28	The constraints for supplementary drivers and delays are clearly stated.	Section 8.11
29	Driver Type function: There should be a driver access function for finding type of driver. driver_type_function ::= \$driver_type(signal_name, signal_index)	Section 8.11.4, Annex D
30	Clarification on the MS synchronization algorithm: Includes a more detailed explanation on the analog-digital synchronization mechanism.	Section 9
31	Truncation versus Rounding mechanism for converting from analog to digital times.	Section 9.3.2.3
32	Spelling mistake on “boltzmann” and “planck” in constants file	Annex D
33	Units for charge, angle and other definitions in disciplines.vams have been changed to adhere to SI standards.	Annex D
34	Values specified in constants file for charge, light, boltzman constant, and so forth have been changed to adhere to the standard definitions.	Annex D

C.19 Obsolete functionality

The following statements are not supported in the current version of Verilog-AMS HDL; they are only noted for backward compatibility.

C.19.1 Forever

This statement is no longer supported.

C.19.2 NULL

This statement is no longer supported. Certain functions such as case, conditionals and the event statement do allow null statements as defined by the syntax.

C.19.3 Generate

The *generate statement* is a looping construct which is unrolled at elaboration time. It is the only looping statement that can contain analog operators. The syntax of generate statement is shown in Figure C.1.

```

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

start_expr ::=
    constant_expression

end_expr ::=
    constant_expression

incr_expr ::=
    constant_expression

```

Figure C.1—Syntax for generate statement

The index shall 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 the 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 (2x) the increment. This pattern is repeated until the lower bound plus a multiple of the increment is greater than the upper bound.

Examples:

This module implements a continuously running (unclocked) analog-to-digital converter.

```

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

```

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


Annex D

Standard definitions

This annex contains the standard definition packages (`disciplines.vams` and `constants.vams`) for Verilog-AMS HDL.

D.1 The `disciplines.vams` file

```

`ifdef DISCIPLINES_VAMS
`else
`define DISCIPLINES_VAMS 1

//
// Natures and Disciplines
//

discipline logic
    domain discrete;
enddiscipline

/*
 * Default absolute tolerances may be overridden 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 Kelvin
nature Temperature
    units      = "K";
    access     = Temp;
`ifdef TEMPERATURE_ABSTOL
    abstol     = `TEMPERATURE_ABSTOL;
`else
    abstol     = 1e-4;
`endif
endnature

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

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

```

```

// Kinematic
// Position in meters
nature Position
    units          = "m";
    access         = Pos;
    ddt_nature     = Velocity;
`ifdef POSITION_ABSTOL
    abstol         = `POSITION_ABSTOL;
`else
    abstol         = 1e-6;
`endif
endnature

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

```

```

// Angular acceleration in radians per second squared
nature Angular_Acceleration
    units      = "rads/s^2";
    access     = Alpha;
    idt_nature = Angular_Velocity;
`ifdef ANGULAR_ACCELERATION_ABSTOL
    abstol     = `ANGULAR_ACCELERATION_ABSTOL;
`else
    abstol     = 1e-6;
`endif
endnature

// Torque in Newtons
nature Angular_Force
    units      = "N*m";
    access     = Tau;
`ifdef ANGULAR_FORCE_ABSTOL
    abstol     = `ANGULAR_FORCE_ABSTOL;
`else
    abstol     = 1e-6;
`endif
endnature

// Conservative disciplines
discipline rotational
    potential   Angle;
    flow        Angular_Force;
enddiscipline

discipline rotational_omega
    potential   Angular_Velocity;
    flow        Angular_Force;
enddiscipline
`endif

```

D.2 The constants.vams file

```
// Mathematical and physical constants

`ifdef CONSTANTS_VAMS
`else
`define CONSTANTS_VAMS 1

// M_ is a mathematical constant

`define      M_E                2.7182818284590452354
`define      M_LOG2E            1.4426950408889634074
`define      M_LOG10E           0.43429448190325182765
`define      M_LN2              0.69314718055994530942
`define      M_LN10             2.30258509299404568402
`define      M_PI               3.14159265358979323846
`define      M_TWO_PI           6.28318530717958647652
`define      M_PI_2             1.57079632679489661923
`define      M_PI_4             0.78539816339744830962
`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

// The following constants have been taken from http://physics.nist.gov

// P_ is a physical constant

// charge of electron in coulombs
`define      P_Q                1.602176462e-19

// speed of light in vacuum in meters/sec
`define      P_C                2.99792458e8

// Boltzmann's constant in joules/kelvin
`define      P_K                1.3806503e-23

// Planck's constant in joules*sec
`define      P_K                6.62606876e-34

// permittivity of vacuum in farads/meter
`define      P_EPS0             8.854187817e-12

// permeability of vacuum in henrys/meter
`define      P_U0               (4.0e-7 * `M_PI) (12.566370614e-7)

// zero celsius in kelvin
`define      P_CELSIUS0        273.15

`endif
```

D.3 The `driver_access.vams` file

```

`define DRIVER_UNKNOWN      32'b000000000000// No information
`define DRIVER_DELAYED     32'b000000000001// driver has fixed delay
`define DRIVER_GATE        32'b000000000010// driver is a primitive
`define DRIVER_UDP         32'b000000000100// driver is a user defined primitive
`define DRIVER_ASSIGN      32'b000000001000// driver is a continuous assignment
`define DRIVER_BEHAVIORAL  32'b000000010000// driver is a reg
`define DRIVER_SDF         32'b000000100000// driver is from backannotated code
`define DRIVER_NODELETE    32'b000001000000// events won't be deleted1
`define DRIVER_NOPREEMPT   32'b000010000000// events won't be preempted
`define DRIVER_KERNEL      32'b001000000000 // added by kernel (wor/wand)
`define DRIVER_WOR         32'b010000000000// driver is on a wor net
`define DRIVER_WAND        32'b100000000000// driver is on a wand net

```

1. For optimization

Annex E

SPICE compatibility

E.1 Introduction

Analog simulation has long been performed with SPICE and SPICE-like simulators. As such, there is a huge legacy of SPICE netlists. In addition, SPICE provides a rich set of predefined models and it is considered neither practical nor desirable to convert these models into a Verilog-AMS HDL behavioral description. In order for Verilog-AMS HDL to be embraced by the analog design community, it is important Verilog-AMS HDL provide an appropriate degree of SPICE compatibility. This annex describes the degree of compatibility which Verilog-AMS HDL provides and the approach taken to provide that compatibility.

E.1.1 Scope of compatibility

SPICE is not a single language, but rather is a family of related languages. The first widely used version of SPICE was SPICE2g6 from the University of California at Berkeley. However, SPICE has been enhanced and distributed by many different companies, each of which has added their own extensions to the language and models. As a result, there is a great deal of incompatibility even among the SPICE languages themselves.

Verilog-AMS HDL makes no judgement as to which of the various SPICE languages should be supported. Instead, it states if a simulator which supports Verilog-AMS HDL is also able to read SPICE netlists of a particular flavor, then certain objects defined in that flavor of SPICE netlist can be referenced from within a Verilog-AMS HDL structural description. In particular, SPICE models and subcircuits can be instantiated within a Verilog-AMS HDL module. This is also true for any SPICE primitives which are built into the simulator.

E.1.2 Degree of incompatibility

There are four primary areas of incompatibility between versions of SPICE simulators.

1. The version of the SPICE language accepted by various simulators is different and to some degree proprietary. This issue is not addressed by Verilog-AMS HDL. So whether a particular Verilog-AMS simulator is SPICE compatible, and with which particular variant of SPICE it is compatible, is solely determined by the authors of the simulator.

2. Not all SPICE simulators support the same set of component primitives. Thus, a particular SPICE netlist can reference a primitive which is unsupported. Verilog-AMS HDL offers no alternative in this case other than the possibility that if the model equations are known, the primitive can be rewritten as a module.
3. The names of the built-in SPICE primitives, their parameters, or their ports can differ from simulator to simulator. This is particularly true because many primitives, parameters, and ports are unnamed in SPICE. When instantiating SPICE primitives in Verilog-AMS HDL, the primitives shall, and parameters and ports can, be named. Since there are no established standard names, there is a high likelihood of incompatibility cropping up in these names.

To reduce this, a list of what names shall be used for the more common components is shown in Annex E.3. However, it is not possible to anticipate all SPICE primitives and parameters which could be supported; so different implementations can end up using different names. This level of incompatibility can be overcome by using wrapper modules to map names.

4. The mathematical description of the built-in primitives can differ. As with the netlist syntax, incompatible enhancements of the models have crept in through the years. Again, Verilog-AMS HDL offers no solution in this case other than the possibility that if the model equations are known, the primitive can be rewritten as a module.

E.2 Accessing SPICE objects from Verilog-AMS HDL

If an implementation of a Verilog-AMS tool supports SPICE compatibility, it is expected to provide the basic set of SPICE primitives (see Annex E.3) and be able to read SPICE netlists which contain models and subcircuit statements.

SPICE primitives built into the simulator are treated in the same manner in Verilog-AMS HDL as built-in primitives. However, while the Verilog-AMS HDL built-in primitives are standardized, the SPICE primitives are not. All aspects of SPICE primitives are implementation dependent.

In addition to SPICE primitives, it is also possible to access subcircuits and models defined within SPICE netlists. The subcircuits and models contained within the SPICE netlist are treated as module definitions.

E.2.1 Case sensitivity

SPICE netlists are case insensitive, whereas Verilog-AMS HDL descriptions are case-sensitive. From within Verilog-AMS HDL, a mixed-case name matches the same name with an identical case (if one is defined in a Verilog-AMS HDL description). However,

if no exact match is found, the mixed-case name shall match the same name defined within SPICE regardless of the case.

E.2.2 Examples

This subsection shows some examples.

E.2.2.1 Accessing SPICE models

Consider the following SPICE model file being read by a Verilog-AMS HDL simulator.

```
.MODEL VERTNPN NPN BF=80 IS=1E-18 RB=100 VAF=50
+ CJE=3PF CJC=2PF CJS=2PF TF=0.3NS TR=6NS
```

This model can be instantiated in a Verilog-AMS HDL module as shown in Figure E.1.

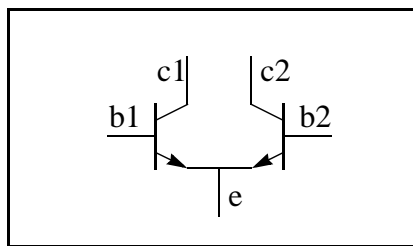


Figure E.1—Instantiated module

```
module diffPair (c1, b1, e, b2, c2);
    electrical c1, b1, e, b2, c2;

    vertNPN Q1 (c1, b1, e, );
    vertNPN Q2 (.c(c2), .b(b2), .e(e));

endmodule
```

Unlike with SPICE, the first letter of the instance name, in this case Q1 and Q2, is not constrained by the primitive type. For example, they can just as easily be T1 and T2.

The ports and parameters of the BJT are determined by the BJT primitive itself and not by the model statement for the BJT. See Annex E.3 for more details. The BJT has 3 mandatory ports (collector, base, and emitter) and one optional port (the substrate). In the instantiation of Q1, the ports are passed by order. With Q2, the ports are passed by name. In both cases, the optional substrate port is defaulted by simply not giving it.

E.2.2.2 Accessing SPICE subcircuits

As an example of how a SPICE subcircuit is referenced from Verilog-AMS HDL, consider the following SPICE subcircuit definition of an oscillator.

```

.SUBCKT ECPOSC (OUT GND)
  VA VCC GND 5
  IEE E GND 1MA
  Q1 VCC B1 E VCC VERTNPN
  Q2 OUT B2 E OUT VERTNPN
  L1 VCC OUT 1UH
  C1 VCC OUT 1P IC=1
  C2 OUT B1 272.7PF
  C3 B1 GND 3NF
  R1 B1 GND 10K
  C4 B2 GND 3NF
  R2 B2 GND 10K
.ENDS ECPOSC

```

This oscillator can be referenced from Verilog-AMS HDL as:

```

module osc (out, gnd);
  electrical out, gnd;
  ecpOsc Osc1 (out, gnd);
endmodule

```

Note: In Verilog-AMS HDL the name of the subcircuit instance is not constrained to start with x as it is in SPICE.

E.2.2.3 Accessing SPICE primitives

To show how various SPICE primitives can be accessed from Verilog-AMS HDL, the subcircuit in Figure E.1 is translated to native Verilog-AMS HDL.

```

module ecpOsc (out, gnd);
  electrical out, gnd;

  vsine #(.dc(5)) Vcc (vcc, gnd);
  isine #(.dc(1m)) Iee (e, gnd);
  vertnnpn Q1 (vcc, b1, e, vcc);
  vertnnpn Q2 (out, b2, e, out);
  inductor #(.l(1u)) L1 (vcc, out);
  capacitor #(.c(1p), .ic(1)) C1 (vcc, out);
  capacitor #(.c(272.7p)) C2 (out, b1);
  capacitor #(.c(3n)) C3 (b1, gnd);
  resistor #(.r(10k)) R1 (b1, gnd);
  capacitor #(.c(3n)) C4 (b2, gnd);
  resistor #(.r(10k)) R2 (b2, gnd);
endmodule

```

E.3 Preferred primitive, parameter, and port names

Table E.1 shows the required names for primitives, parameters, and ports which are otherwise unnamed in SPICE. For connection by order instead of by name, the ports and

parameters shall be given in the order listed. The default discipline of the ports for these primitives shall be `electrical` and their descriptions shall be `inout`.

Table E.1—Required names

Primitive name	Port name	Parameter name
resistor	p, n	r, tc1, tc2
capacitor	p, n	c, ic
inductor	p, n	l, ic
vexp	p, n	dc, mag, phase, val0, val1, td0, tau0, td1, tau1
vpulse	p, n	dc, mag, phase, val0, val1, td, rise, fall, width, period
vpwl	p, n	dc, mag, phase, wave
vsine	p, n	dc, mag, phase, offset, ampl, freq, td, damp, sinephase, ammodindex, ammodfreq, ammodphase, fmmodindex, fmmodfreq
iexp	p, n	dc, mag, phase, val0, val1, td0, tau0, td1, tau1
ipulse	p, n	dc, mag, phase, val0, val1, td, rise, fall, width, period
ipwl	p, n	dc, mag, phase, wave
isine	p, n	dc, mag, phase, offset, ampl, freq, td, damp, sinephase, ammodindex, ammodfreq, ammodphase, fmmodindex, fmmodfreq
diode ^a	a, c	area
bjt ^a	c, b, e, s	area
mosfet ^a	d, g, s, b	w, l, ad, as, pd, ps, nrd, nrs
jfet ^a	d, g, s	area
mesfet ^a	d, g, s	area
vcvs	p, n, ps, ns	gain
vccs	sink, src, ps, ns	gm
tline	t1, b1, t2, b2	z0, td, f, nl

a. The names `diode`, `bjt`, `jfet`, `mesfet`, and `mosfet` are never used from within Verilog-AMS HDL because these components require a model. Thus, the model name is used in Verilog-AMS HDL instead of the primitive name.

E.3.1 Independent sources

The parameters associated with each type of independent source are given in Table E.2. “ac” and “dc” parameters are common to each source type and need to be specified to list parameters by order before any waveshape parameters are specified.

Table E.2—Independent source parameters

Source Type	Parameter	Description
“ac” ALL	mag, phase	AC small signal level and phase
“dc” ALL	dc	DC value
“pulse”	val0, val1	Pulse levels
	td	Start time of first pulse
	rise, fall	Pulse rise and fall time
	width	Pulse width
	period	Pulse period

Table E.2—Independent source parameters, *continued*

Source Type	Parameter	Description
"pwl"	wave	Vector of time/value pairs defining the waveform
"sine"	offset	DC level of sinusoid
	ampl	Amplitude of sinusoid
	freq	Frequency of sinusoid
	td	Start time of first pulse
	damp	Damping factor of sinusoid
	sinephase	Phase of sinusoid
	ammodindex	AM index of modulation
	ammodfreq	AM modulation frequency
	ammodphase	AM modulation phase
	fmmodindex	FM index of modulation
	fmmodfreq	FM modulation frequency
"exp"	val0, val1	Equilibrium levels
	td0, td1	Start time for transitions to val0, val1
	tau0, tau1	Time constant for transition to val0, val1

E.3.2 Unsupported components

Verilog-AMS HDL does not support the concept of passing an instance name as a parameter. As such, the following components are not supported: `ccvs`, `cccs`, and mutual inductors; however, these primitives can be instantiated inside a subcircuit.

E.3.3 Discipline of primitives

To afford the ability to use analog primitive in any design, including mixed disciplines, the default discipline override is provided. The discipline of analog primitives will be resolved based on instance specific attributes, the disciplines of other instances on the same net, or default to electrical if it cannot be determined.

The precedence for the discipline of analog primitives is as follows:

1. A `port_discipline` attribute on the analog primitive
2. The resolution of the discipline
3. The default analog primitive of electrical

E.3.3.1 Setting the discipline of analog primitives

A new optional attribute shall be provided called "`port_discipline`", which shall have as a value the desired discipline for the analog primitive. It shall only apply to the instance to which it is attached. The value shall be of type string and the value must be a valid discipline of domain continuous. This attribute shall only apply to analog primitives and for all other modules shall be ignored. The following provides an example of this attribute.

```

resistor #(.r(1k)) (* integer port_discipline="electrical" ; *) r1
(node1, node2); // not needed as default

resistor #(.r(1k)) (* integer port_discipline="rotational" ; *) r2
(node1, node2);

```

Attributes are defined in the IEEE 1364-2001 LRM and will not be described in this document.

E.3.3.2 Resolving the disciplines of analog primitives

If no attribute exists on the instance of an analog primitive, then the discipline may be determined by the disciplines of other instances connected to the same net segment. The disciplines of the vpiLoConn of all other instances on the net segment shall be evaluated to determine if they are of domain continuous and compatible with each other. If they are, then the discipline of the analog primitive shall be set to the same discipline. If they are not compatible, then an error will occur as defined in Section 3.8 of this LRM. If there are no continuous disciplines defined on the net segment, then the discipline shall default to electrical.

E.4 Other issues

This section highlights some other issues

E.4.1 Multiplicity factor on subcircuits

Some SPICE simulators support a multiplicity factor (*M*) parameter on subcircuits without the parameter being explicitly being declared. This factor is typically used to indicate the subcircuit should be modeled as if there are a specified number of copies in parallel. If supported by the implementation, the automatic *M* factors are supported for subcircuits defined in SPICE, but not for subcircuits defined as a modules in Verilog-AMS HDL. Thus, if the SPICE subcircuit in Annex E.2.2.2 was instantiated, a multiplicity factor could be specified (assuming the simulator implementation supports multiplicity factors on SPICE subcircuits. However, a multiplicity factor can not be specified when instantiating the equivalent Verilog-AMS HDL module shown in Annex E.2.2.3.

E.4.2 Binning and libraries

Some SPICE netlists provide mechanisms for mapping an instance to a group of models, with the final determination of which model to use being based on rules encapsulated in the SPICE netlist. Examples include model binning or corners support. From within an instance statement, it appears as if the instance is referencing a simple SPICE model; supporting these additional capabilities in Verilog-AMS HDL is supported via the

instance line by default. Support of model cards are implementation specific (including those using these mechanisms).

Annex F

Discipline resolution methods

F.1 Discipline resolution

Discipline resolution is described in Section 8.4; it provides the semantics for two methods of resolving the discipline of undeclared interconnect. This annex provides a possible algorithm for achieving the semantics of each method. It is also possible to develop and use other algorithms to match the semantics.

F.2 Resolution of mixed signals

The following algorithms for discipline resolution of undeclared nets provide users with the ability to control the auto-insertion of connection modules. The undeclared nets are resolved at each level of the hierarchy in which *continuous* (analog) has precedence over *discrete* (digital). In both algorithms, the *continuous* domain is passed up the hierarchy from lower levels to the top level.

F.2.1 Default discipline resolution algorithm

This default algorithm propagates both continuous and discrete disciplines up the hierarchy to meet one another. Insertion of interface elements shall occur at each level of the hierarchy where both continuous and discrete disciplines meet. This results in connection modules being inserted higher up the design hierarchy. The algorithm is described as follows.

1. Elaborate the design

After this step, every port in the design has both its upper (actual) connection and its lower (formal) connection defined.

2. Apply all in-context node and signal declarations

For example, `electrical sig;` makes all instances of `sig` electrical, unless they have been overridden by an out-of-context declaration.

3. Apply all out-of-context node and signal declarations.

For example, `electrical top.middle.bottom.sig;` overrides any discipline which may be declared for `sig` in the module where `sig` was declared.

More than one conflicting in-context discipline declaration or more than one conflicting out-of-context discipline declaration for the same hierarchical segment of a signal is an error. In this case, *conflicting* simply means an attempt to declare more than one discipline regardless of whether the disciplines are compatible or not.

4. Traverse each signal hierarchically (depth-first) when a net is encountered which still has not been assigned a discipline:
 - A. It shall be determined whether the net is analog or digital. Any net whose port connections (i.e., connections to the upper part of a port) are all digital shall be considered digital (discrete domain), any others shall be considered analog (continuous domain).
 - B. Apply any ``default_discipline` directives to any net segments which do not yet have a discipline, provided their domain is the same as the domain of the default discipline. This is done according to the rules of precedence for ``default_discipline` (see Section 3.7).
 - C. If the segment has not yet been assigned a discipline, examine all ports to which the segment forms the upper connection and construct a list of all disciplines at the lower connections of these ports whose domains match the domain of the segment:
 - If there is only a single discipline in the list, the signal is of that discipline
 - If there is more than one discipline in the list and the contents of the list match the discipline list of a resolution connect statement, the net is of the resolved discipline given by the statement.
 - Otherwise the discipline is unknown. This is legal provided the net segment has no mixed-port connections (i.e., it does not connect through a port to a segment of a different domain). Otherwise this is an error.

At this point, connection module selection and insertion can be performed. Insert converters applying the rules and semantics of the connect statement (Section 8.7) and auto-insertion sections (Section 8.8).

F.2.2 Alternate expanded analog discipline resolution algorithm

This algorithm propagates continuous disciplines up and then back down to meet discrete disciplines. This may result in more connection modules being inserted lower down into discrete sections of the design hierarchy for added accuracy. The selection of this algorithm instead of the default shall be controlled by a simulator option. The algorithm is described as follows.

1. Elaborate the design

After this step, every port in the design has both its upper (actual) connection and its lower (formal) connection defined.

2. Apply all in-context node and signal declarations

For example, `electrical sig;` makes all instances of `sig` electrical, unless they have been overridden by an out-of-context declaration.

3. Apply all out-of-context node and signal declarations.

For example, `electrical top.middle.bottom.sig;` overrides any discipline which may be declared for `sig` in the module where `sig` was declared.

More than one conflicting in-context discipline declaration or more than one conflicting out-of-context discipline declaration for the same hierarchical segment of a signal is an error. In this case, *conflicting* simply means an attempt to declare more than one discipline regardless of whether the disciplines are compatible or not.

4. Traverse each signal hierarchically (depth-first) when a net is encountered which has still not been assigned a discipline:
 - A. It shall be determined whether the net is analog or digital. Any net whose port connections (i.e., connections to the upper part of a port) are all digital shall be considered digital. If any of the connections are analog, the net shall be considered analog. Any others shall still be considered unknown.
 - B. Apply any ``default_discipline` directives to any net segments which do not yet have a discipline, provided their domain is the same as the domain of the default discipline. This is done according to the rules of precedence for ``default_discipline` (see Section 3.7).
 - C. If the segment has not yet been assigned a discipline, examine all ports to which the segment forms the upper or lower connection. and construct a list of all disciplines at the other connections of these ports whose domains match the domain of the segment:
 - If there is only a single discipline in the list, the signal is of that discipline
 - If there is more than one discipline in the list and the contents of the list match the discipline list of a resolution connect statement, the net is of the resolved discipline given by the statement.
 - Otherwise the discipline is unknown.

5. Traverse each net hierarchically (top-down) when a net is encountered which still has not been assigned a discipline:
 - A. It shall be determined whether the net is analog or digital. Any net whose port (i.e., connection to the lower part of a port) is digital shall be considered digital. Any others shall be considered analog.
 - B. Apply any ``default_discipline` directives which to any net segments which do not yet have a discipline, provided their domain is the same as the domain of the default discipline. This is done according to the rules of precedence for ``default_discipline` (see Section 3.7).
 - C. If the segment has not yet been assigned a discipline, examine all ports to which the segment forms the lower connection and construct a list of all disciplines at the upper connections of these ports whose domains match the domain of the segment:
 - If there is only a single discipline in the list, the signal is of that discipline
 - If there is more than one discipline in the list and the contents of the list match the discipline list of a resolution connect statement, the net is of the resolved discipline given by the statement.
 - Otherwise the discipline is unknown. This is legal provided the net segment has no mixed-port connections (i.e., it does not connect through a port to a segment of a different domain). Otherwise this is an error.

At this point, connection module selection and insertion can be performed. Insert converters applying the rules and semantics of the connect statement (Section 8.7) and auto-insertion sections (Section 8.8).

Annex G

Open issues

Annex G identifies issues in this LRM version that require additional investigation, and which will be addressed in the next revision. The following table describes the issues and categorizes them as follows:

- **Ambiguity**
Indicates something that is unclear in the current version of the LRM
- **BNF**
Indicates that the Verilog-AMS syntax grammar requires modification
- **Enhancement**
Indicates something that is not addressed in the current language but will be addressed in future revisions

Table G.1—Open issues in LRM 2.1

Item	Description/Issue	Category
1	wreal declaration and usage should be synchronized with System Verilog language. The BNF allows a wreal declaration without any identifier, for example, wreal; Change syntax from wreal_declaration := wreal [list_of_identifiers]; to wreal_declaration := wreal list_of_identifiers;	BNF
2	Verilog-AMS has no language extensions to support back annotation.	Enhancement
3	Discipline Resolution Issues: Algorithm is based on net types rather than driver that appears on mixed net. Related Issues: <ul style="list-style-type: none"> • Driver-Receiver segregation • placement of A/D converter • empty disciplines, undeclared nets • how to deal with leaf level wires • no clear definition on OOMR • What is the discipline of an expression when an expression is specified as an actual in a port connection? A driver might have to be created as an unnamed implicit function. 	Ambiguity

Item	Description/Issue	Category
4	<p>LRM does not clearly illustrate the MS simulation cycle and the initialization is not clearly defined. Illustration of IC analysis in AMS is non-existent.</p> <ul style="list-style-type: none"> • Which solver starts first • Initialization mechanism • Rules for synchronization are not clear enough and could impact portability <p>Initialization with digital engine:</p> <ul style="list-style-type: none"> • Mixed-signal initialization: Verilog-D simulators are transient in operation and hence there is no mechanism defined for static/steady state simulation. 	Ambiguity
5	System tasks and function. Issue with \$random . The syntax is defined in both digital (1364-2001) as well as analog (LRM 2.1) BNF. The syntax should be in sync.	BNF
6	Issue with genvar. Currently genvar is defined both in Verilog-D 1364-2001 std as well as LRM 2.1, and they are used in different contexts. Digital genvar is used more in structural context and analog genvar is used in a behavioural (<i>analog_for</i>) context.	BNF
7	<ul style="list-style-type: none"> • External module definition to support direct import of SPICE netlists in Annex E. Currently this is not clearly stated in the LRM. • Augment “external module” and “macromodule” definitions within a “simulator class”. Extend the “external module” syntax with parameters indicating the language used. Also, case sensitivity of SPICE simulators should be accounted for in the language. Should there be filters for support of foreign language? Implement something similar to “shell” which is used in digital simulators. 	Enhancement
8	Support for global variables using dynamic parameters	Enhancement
9	Ambiguities with <i>if-else-if</i> BNF syntax, when nested <i>if</i> statements are used, when the <i>if-condition</i> is statically evaluable.	BNF
10	<p>Contribution statements:</p> <ul style="list-style-type: none"> • It is not clear whether contribution statements should be allowed as part of initial conditions like initial_step. The current BNF restricts the use of contribution statements in this context. • Contribution statement inside loops. Should this be allowed? Current LRM disallows using contribution statements in loops. 	Enhancement
11	<p>Analysis-dependent function should be clearly defined with use of tables to denote how they behave.</p> <p>This should also clarify DC Sweep mechanism, which is not detailed in the LRM. Behaviour of initial/final step on DC Sweep is also not defined.</p>	Ambiguity
12	Syntax consistencies with 1364 in BNF snippets specified while describing the feature Syntax 6-3, 6-4, 6-5 in Section 6.3, 6.4, and 6.5 are inconsistent with the BNF syntax defined in Annexure. The syntax snippets in the actual sections should be the same as the one specified in Annex.	BNF
13	SPICE versus Verilog name conflict. There is an issue while instantiating two modules with the same name defined in different abstractions (SPICE versus Verilog). Currently, the LRM does not have any name scoping mechanisms. Look at the possibility of issuing an error when a name conflict occurs, or use the standard library methodology to pick the first match from the library.	Enhancement
14	Switch branch syntax not defined in BNF, though it is explained in an example. Also, should indirect branch assignment be made illegal in conditional? This is not clearly defined in the LRM (direct contribution statements are allowed in <i>if</i> statements to model switch behavior).	BNF

Item	Description/Issue	Category
15	Discipline Compatibility: How do you resolve disciplines with different abstol. Should there be a resolution function? Should the tighter tolerance apply?	Ambiguity
16	Discipline Compatibility: Current LRM makes it illegal to have incompatible continuous disciplines on the same net. Should this be allowed?	Enhancement
17	The meaning of ‘analysis point’ in the table explaining initial_step/final_step for all analysis is not clear (Section 6.7.4)	Ambiguity
18	‘include does not support both <> and “” to include header files. This syntax could be used to differentiate system-defined header files and user-defined header files.	Enhancement
19	The specification of roots for the Zi filter has changed from Z^{-1} to Z. Poles and zeros are roots of Z^{-1} , but this changed in post 1.4 versions.	Ambiguity
20	Should flow and potential etc. be part of global keyword list? Should there be a single global keyword list?	Enhancement
21	Coercion of string to real does not make sense and is not defined, but it is currently allowed. The LRM should not allow coercing of a string to real . This should be treated as an error in the analog context.	BNF
22	The concept of associating variables based on which context they are assigned is apparently felt to be ambiguous. Can there be a better approach to identifying the domain of a variable in a mixed-signal context?	Enhancement
23	Can “,” (comma-comma) syntax be used for null arguments instead of “{}”? This can be useful for defining NULL numerator/denominator arguments in Laplace/Zi. Can this be explicitly allowed for these operators?	Enhancement
24	How will we handle parameter override values specified in defparam versus instantiation precedence? If the module is pre-compiled, the compiler would not know about the last override value. (Apparently there was a discussion on Verilog-D not to support defparam .)	Enhancement
25	Light Weight Conversions: connectmodule only addresses automatic conversion of signals passed through ports (structural). What about OOMRs where behavioral code asks for values not passed through ports? These are just probes, which do not need resolution. There should be a mechanism in the LRM for treating these light weight conversions.	Enhancement
26	VPI issue: Nature and disciplines should not use param_assign, instead there should be a new object called attr_assign	Ambiguity
27	Support break/continue statement in behavioral syntax to break out of loops.	Enhancement
28	Indexing of Named vector branches: What should be the index of a vector branch declared between two vector nodes: <pre>electrical [1:2] a; electrical [0:1] b; branch (a,b) br1 [1:2]; branch (a,b) br2; //??</pre> Should br2 default to [0:1]?	BNF

Item	Description/Issue	Category
29	Bitwise nand/nor is mentioned in the operators precedence table, but there is no mention of these operators in the list of bitwise operators	BNF
30	Currently, the digital syntax does not allow register declaration in named blocks mentioned in an @(initial) or @(always) block. Register and net declarations should be allowed as part of these named blocks	Enhancement
31	Vector return values for UDF: LRM do not explicitly state that the return value of UDF should be scalar. This should be clarified.	Ambiguity
32	Events inside loop statements: Events are used inside loops in references through examples (Section 5.2, dac), though this is disallowed as per the grammar.	Ambiguity
33	Accessing discipline attributes (both digital and analog) to be supported in the language.	Enhancement

Annex H

Glossary

Glossary of Terms

A

AMS

See also, *Verilog-AMS*.

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 which encapsulates behavior and/or structure. Modules and models can represent a single component or a subcircuit with many components.

constitutive relationships

The essential relationships (*expressions* and *statements*) between the outputs of a module and its inputs and parameters, which 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 another, “parent” module. A complete definition of the child module needs to be defined somewhere. A child module is also known as *submodule* or *instantiated module*.

F

flow

One of the two fundamental quantities used to simulate the behavior of a system. In electrical systems, the flow is the current.

I

instance

Any named occurrence of an component created from a module definition. One module 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

The physical laws defining 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** or **discipline-endsdiscipline**.

M

model

A named instance with a unique group of parameters specifying the behavior of one particular version of a module. Models can be used to instantiate elements with parametric specifications different from those in the original module definition.

module

A definition of the interfaces and behavior of a component.

N

nesting level

One block within a behavioral description, delimited by a pair of matching keywords such as **begin-end** or **discipline-endsdiscipline**.

node

A connection point in the system, with access functions for potential and/or flow through an underlying discipline.

node declaration

The statement in a module definition identifying the names of the nets associated with the module ports or local to the module. A net declaration also identifies the discipline of the net, which in turn identifies the access functions.

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.

P

parameter

A constant 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 parameters of that module.

port

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. In electrical systems, the potential is the voltage.

primitive

A basic component defined entirely in terms of behavior, without reference to any other primitives. A primitive is the smallest and simplest portion of a simulated circuit or system.

probe

A branch in a circuit (or system), which does not alter its behavior, but lets the simulator 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 (0)) against whose potentials all node values are measured. Nets declared as `ground` shall be bound to the reference node.

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

See also, *port*.

V

Verilog-A

A subset of Verilog-AMS detailing the analog version of Verilog HDL (see Annex C). This is a language for the behavioral description of continuous-time systems, which uses a syntax similar to the *IEEE 1364-2001 Verilog HDL* specification.

Verilog-AMS

Mixed-signal version of Verilog HDL. A language for the behavioral description of continuous-time and discrete-time systems based on the *IEEE 1364-2001 Verilog HDL* specification.

