

Spectre AMS Designer and Xcelium Simulator Mixed-Signal User Guide

**Product Version 22.09
September 2022**

© 2023 Cadence Design Systems, Inc. All rights reserved worldwide.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders. Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. Open SystemC, Open SystemC Initiative, OSCI, SystemC, and Accellera Systems Initiative Inc. are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission. All other trademarks are the property of their respective owners.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement permits Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly outlined in such agreement, Cadence does not make, and expressly disclaims any representations or warranties as to the completeness, accuracy, or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third-party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from the use of such information.

Cadence is committed to using respectful language in our code and communications. We are also active in the removal and/or replacement of inappropriate language from existing content. This product documentation may however contain material that is no longer considered appropriate but still reflects long-standing industry terminology. Such content will be addressed at a time when the related software can be updated without end-user impact.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1	20
Product and Licensing Information for Mixed-Signal Designs	20
Related Topics	22
Platform Support	22
License Checkout Order for Mixed-Signal Designs	24
License Requirement Report	25
Feature-Specific License Checkout Order	27
Typographic and Syntax Conventions	32
2	34
Spectre AMS Designer Simulator and Xcelium Simulator with Mixed-Signal Option Capabilities	34
Related Topics	34
3	36
AMS Designer Simulator	36
Related Topics	36
Language Support	36
Versions of the Spectre AMS Designer Simulator	37
Setting the Path Variable to Point to the 64-Bit Version	37
Library Path Environment Variable	37
Running the AMS Designer Simulator	37
Single-step Approach to Running a Simulator	38
AMS Designer Flex Mode	38
Analog Solver Information	38
Analog Solvers Supported by AMS Designer Simulator	39
Advanced Analog Solver Features	39
Compatible Spectre Version Control File	39
The spectre_compatible_version.xml File Location	40
4	42
Analog Simulation Control File	42
Related Topics	42
Switching Languages in the Analog Simulation Control File	42

Using Spectre lookup Rules	43
Spectre Language Statements in the Analog Simulation Control File	44
Passing the Analog Simulation Control File to xrun	44
Passing the Analog Simulation Control File to xmsim	45
5	46
Controls for the Analog Solvers	46
Related Topics	46
Spectre AMS Designer Analysis Support	46
Immediate Set Options (options)	46
AC Analysis Statement (ac)	49
Noise Analysis (noise)	50
Transient Analysis Statement (tran)	51
Parameters	51
Examples	53
Monte Carlo Analysis	54
EMIR Analysis in Spectre AMS Designer	54
Reliability Analysis in Spectre AMS Designer	55
Checklimit Analysis in Spectre AMS Designer	55
DC Analysis in Spectre AMS Designer	56
fastdc Option	56
dcopt Option	56
Sweep Analysis (sweep)	57
Stability Analysis (stb)	57
Transfer Function Analysis (xf)	57
Initial Conditions (ic) Statement	58
alter Statement	58
Alter Group (altergroup)	58
Device Checking and Violations Display	59
info Statement	59
what	59
where	60
file	60
extremes	60
save	60
title	61
writedc	61

Examples	61
save Statement	61
print Statement	62
Mixed-Signal DC Initialization	62
Comparing the Behavior of save Parameter Specified in an Analysis and the Global save Option	63
Circuit Checks for Spectre AMS Designer	63
Fault Analysis in Spectre AMS Designer	66
Fault Simulation Use Model	66
AMS Designer Spectre Fault Analysis Integration	68
nodeset Statement	68
nodeset Statement in Analog Simulation Control File	69
Spectre or APS Block-Based Transient Noise Analysis	69
State Information for Individual Analyses	70
Time-Saving Techniques for the Analog Solvers	70
Adjusting Speed and Accuracy	70
Saving Time by Selecting a Continuation Method	71
Specifying Efficient Starting Points	71
6	73
amsd Block	73
Related Topics	73
amsd Block Statements and Syntax	73
portmap	74
Implicit and Explicit Port Mapping	77
Spice-to-Verilog Port Mapping	78
SPICE	80
config	81
Verilog 2001 Configuration File	83
ie	84
Dynamic Voltage Supply	85
Custom Connect Rules	86
Scope Assignments	87
Parameter Assignments	90
ce	94
connectmap	95
Integration between connectmap and ie Cards	96

Hierarchical Interface Element Optimization	96
Hierarchical Optimization	97
Interface Element Optimization between Dissimilar Nets	101
Discipline Resolution Optimization	103
Xcelium Mixed Signal: Co-Simulation Functionalities	104
Universal Connect Module (UCM)	106
UCM Compatibility with Legacy Connect Rules	106
UCM Use Model	107
7	108
Analog Primitives and Subcircuits	108
Determining the Discipline of Analog Primitive Ports	108
Related Topics	109
Port Bindings	109
Binding Ports by Order	110
Guidelines for Binding Ports by Name or Order	111
Parameter Binding	112
Analog Instances Inside Generate Statements	113
Subcircuits and Models	115
Compiled C Flow	115
Incremental Compilation of C Code	115
Disabling the Compile C Flow	116
Multi-Threading	116
Using mtline in a Schematic with AMS Simulator	116
Using mtline in a SPICE or Spectre Subcircuit or Model	118
8	120
Design Configuration Using Mixed Languages	120
Related Topics	120
Verilog-AMS Connect Modules for VHDL-SPICE Connection	120
Verilog-A Modules in SPICE Blocks	121
Verilog-AMS Vector Buses with SPICE Subcircuits	122
SPICE-on-Top	122
SPICE-in-the-Middle	124
Specifying DSPF and SPEF Stitching on Analog and Mixed-Signal Nets	125
SPICE Blocks in the Middle of VHDL Blocks	127
Using DSPF-in-the-Middle	129
-keepcase4use5x	130

Usage of Port Expressions when Connecting to Analog Blocks	131
SPICE Nets inside a Verilog Design	133
Reusing Mixed-Language Testbenches	138
Using Command-Line Options to Manage Out-of-Module References to SPICE	138
Using Compiler Directives to Manage Out-of-Module References to SPICE	143
Using Bus Delimiters with Out-of-Module References to SPICE	144
Values Associated with an Analog Object	145
Strength-Based Interface Element (SIE)	149
PSL Assertions	152
Limitations of Using PSL Assertions	154
Reusing Testbenches and IP	156
Connecting VHDL Blocks to SPICE Blocks	157
Connections from VHDL and VHDL-AMS Blocks to Verilog and Verilog-AMS Blocks	161
Importing Verilog-AMS Modules into VHDL	164
Importing a Verilog-AMS Module	164
xmshell Command	169
Inherited Connections in VHDL-AMS	170
Mapping Verilog-AMS Disciplines to VHDL-AMS Natures	172
SPICE Built-In Primitives in VHDL-AMS and VHDL-Digital	172
String Type Literals and Generics in VHDL-AMS	175
VHDL-SPICE Conversion Element Optimization	176
CE Information in VHDL-SPICE	177
9	179
Real Number Modeling	179
Basic wreal Features of the AMS Designer Simulator	179
Advanced wreal Features of the AMS Designer Simulator	180
Related Topics	184
Using the wreal Data Type	184
wreal Independent Variables in a \$table_model	185
Verilog-AMS wreal and Analog Signal Connections	188
Resolving Disciplines for Verilog-AMS wreal Nets	191
VHDL Blocks with Real Signal Ports on a Schematic	193
wreal Nets at Mixed-Language Boundaries	194
Writing Bidirectional Model Behavior	194
Using wreal in Assertions	199
Selecting a wreal Resolution Function	199

Order of Precedence for Determining the Resolution Function on a Net	202
Rules to Determine the Resolved Resolution Function for Connected wreal Nets	202
Predefined wreal Resolution Functions	203
Verilog-AMS–Based RNM for Wreals	210
Writing RNMs Using Disciplineless Wreal Nets	211
Writing Portable Verilog-AMS Wreal Models for Use in SV	212
Wreal Concatenation Expressions	212
L2R and R2L Connect Modules	214
Port Connections through Real-to-Logic Connect Modules	217
Inherited Connections in R2L/L2R Connect Modules	220
Aliasing Nets to Hierarchical Nets in L2R/R2L Interface Elements	223
L2R/R2L Insertion in Multiple-Supply Designs	226
Electrical Supply in Real-to-Logic and Logic-to-Real Interface Elements	226
Virtuoso Visualization and Analysis in xrun and ADE Flows for Simulations with Real Number Models	227
Real Number Modeling Debug Options	228
wreal to VHDL Connections	228
10	231
XRUN Invocation for Mixed-Signal Designs	231
Related Topics	231
xrun Command Syntax	231
xrun Command-Line Options for Mixed-Signal Designs	234
Examples Using xrun for AMS Simulation	258
Using xrun with Spectre and SPICE Input Files	258
Migrating to Single-Step XRUN Invocation	261
Mixed-Signal Design Debugging Reports	263
PSpice Netlist and Device Models	266
License Queuing	267
AHDL Linter	268
Turning on Spectre Parasitic Reduction	269
Turning on Spectre Multithreading for Device Evaluation	270
Turning off Spectre Multithreading	270
Simulation Diagnostics	271
Enabling AMS-APS Mode	272
AMS Spectre XPS MS Mode	272
AMS Designer with Spectre FX	277

Options for AMS Designer with Spectre FX	278
Connect Module Usage	278
Loading Plug-In for Spectre Netlist Compiled Functions (NCFs)	279
11	280
AMS Designer Simulator for Design Verification	280
Related Topics	280
Creating a Testbench	280
Run Script for xrun	283
Tcl File Creation to Probe Behavioral Nodes	283
Port Bindings between Verilog and SPICE	284
Binding Ports using autobus	284
Binding Ports using a Port-Bind File	286
Binding Ports using a Verilog File	286
Binding Ports by Name	289
Customized Port-Bind Files	290
Customized Port-Bind File Examples	291
Rules That Apply to Customized Port-Bind Files	292
12	294
Multiple Power Supplies for Designs	294
Related Topics	294
ie Statement in an amsd Block for Multiple Power Supply Design	295
Supply Value to Apply to a Library, Cell, or Instance	295
Cadence-Installed Connect Rules Customization	296
Cadence-Provided Connect Rules	297
Block-Based Discipline Resolution for Multiple Power Supply Design	297
Connect Modules, Connect Rules, and Discipline Definitions	297
Custom Connect Modules, Connect Rules, and Disciplines on the Command Line	299
Supply-Sensitive Modules for Multiple Power Supply Designs	300
Supply-Sensitive Connect Modules	301
Supply-Sensitivity Attributes to an Ordinary Module	304
Interface Elements for Multiple Power Supply Designs	305
13	309
Mixed-Signal Design Elaboration	309
Illustrating the xmelab Process	309
Related Topics	310

Access to Digital Simulation Objects	310
Binding During Mixed-Signal Design Elaboration	311
Delay Modes in Mixed-Signal Designs	312
Guidance for Specifying Disciplines for Scopes	313
Precedence of Discipline Specification Methods	316
Setting Block-Based Discipline Resolution Specifications in Search Libraries	318
Setting Pulse Controls for Mixed-Signal Designs	319
Simulation Front End (SFE) Parser	319
Including Structural Verilog-A in a Spectre Netlist	321
Using SPICE and Spectre User-Defined Functions	322
Using Simplified Input Commands with the Simulation Front End Parser	322
Migrating from the Old Spectre Parser	323
14	324
Mixed-Signal Design Simulation	324
Related Topics	325
Running the Simulator	325
Starting or Resuming a Simulation	326
Restarting the Simulator from a Previously-Saved Snapshot	326
Use Model 1 (Warm Restart)	327
Use Model 2 (Cold Restart)	327
Using the Save-and-Restart Feature	328
Mixed-Signal Activity Statistics	332
Other Tasks During Simulation	335
Exiting the Simulation	336
Specifying a Host for Spectre	337
Managing Analog Resource Usage During Interactive Simulations	337
15	339
SimVision with AMS Simulator	339
The Design Browser Window for AMS Designs	339
Related Topics	341
Invoking SimVision	342
SimVision Window Menu Bar	344
	344
Setting Display Preferences for Verilog-AMS Objects	346
	346
Setting Formatting Preferences for Verilog-AMS Objects	348

Editing Source Information Using the Source Browser	349
Plotting Signals in the Waveform Window	350
The Console Window	351
Cross-Probing Instances and Nets	352
Cross-Probing Instances	353
Cross-Probing Nets	355
Analog Mnemonic Maps in SimVision	358
	358
Real Value Probe Filtering	360
	360
16	362
Debugging Mixed-Signal Designs	362
Terminology	362
Downgrading Mixed-Signal Initialization Errors	363
Forcing and Releasing Signal Values	363
Managing Breakpoints	364
After setting breakpoints, you can display information on breakpoints, disable breakpoints, enable previously disabled breakpoints, and delete breakpoints.	364
Managing Databases	365
Opening a Database	365
Displaying Information about Databases	366
Disabling a Database	366
Enabling a Database	367
Closing a Database	367
Mixed Network Debugging	368
Model Hierarchy for Spectre AMS Designer	368
Paths and Mixed-Language Designs	369
Setting and Deleting Probes	371
Setting a Probe Using the Tcl probe Command	371
Displaying Information about Probes Using the Tcl probe Command	372
Disabling a Probe Using the Tcl probe Command	372
Enabling a Probe Using the Tcl probe Command	372
Deleting a Probe Using the Tcl probe Command	373
Stepping through Lines of Code	373
Breakpoints	374
Setting a Condition Breakpoint	374

Setting a Line Breakpoint	375
Setting a Signal Breakpoint	375
Setting a Time Breakpoint	376
Setting a Process Breakpoint	376
Setting a Subprogram Breakpoint	376
Depositing Values to Signals	377
Displaying Information about Simulation Objects	378
Displaying the Drivers of Signals	378
Automatically-Inserted Connect Modules	379
Waveforms	379
Creating a Database and Probing Signals	380
Opening a Database with \$shm_open	380
Probing Signals with \$shm_probe	381
Opening the SimVision Waveform Window	382
Displaying Debug Settings	383
Setting Variables	384
	384
Editing a Source File Using Your Own Editor	385
Searching the Source Code	386
Searching for a Line Number in the Source Code	386
Searching for a Text String in the Source Code	386
Saving and Restoring Your Simulation Environment	386
Creating or Deleting an Alias	387
	387
Getting a History of Commands	387
Managing Custom Buttons	388
17	389
Xcelium Simulator Mixed-Signal Capabilities	389
18	390
SystemVerilog Real Number Modeling	390
Related Topics	390
Using Real Number Modeling in SystemVerilog	390
Specifying Built-In SystemVerilog Nettypes	391
Specifying User-Defined Nettype and Resolution Function	394
Resolving Wreal Nets of Built-In Nettype	403
Handling Port Connections of Nettypes	404

Wreal Interaction with Built-In Real Nettypes and Real Nettypes with Resolution Functions	406
Wreal Interaction with Nets of Built-In Nettype	407
SystemVerilog Interconnects	408
Port Connection Rules	410
Port Collapsing Rules	411
Interconnect Connections to VHDL Ports	412
Discipline Resolution	412
Net Delays on Interconnects	412
SystemVerilog User Defined Nettype and Electrical Connections	413
Using the Built-In EE Package Connect Modules for UDN-to-Electrical Connections	413
Using Custom User-Defined Nettype and Connect Modules for UDN-to-Electrical Connections	416
Enabling Net Aliasing for SV-UDN Connections	420
Wreal Connections with Concatenated Nettypes to Array of Instances	421
Example 1	421
Example 2	422
Example 3	423
Example 4	423
Limitations	424
Wreal Connections with Concatenated Wires and Nettype Ports	425
19	427
SystemVerilog and AMS Extensions	427
Related Topics	435
SystemVerilog Binding on SPICE	435
Wildcard-Named Port Connections in AMS	436
SystemVerilog Assertions on real, wreal, and electrical Nets	440
SystemVerilog Assertions	440
Analog System Tasks in SVA	442
Using Custom ie Parameter and Connect Module Names for UDN-to-Electrical Connections	443
Configuring SV-AMS Connect Modules for UDN-UDN, UDN-Logic, and UDN-Real Connections	447
Complex Port Definitions with User-Defined Nettypes	454
Using the SVAMS Parser	456
Variable To Wire (VTW) Optimization in Mixed-Signal Designs	457
Features Enabled Using the -adv_ms Option	458
Connecting SystemVerilog Logic Ports to Electrical Using Wire/Interconnects	458

Connecting SystemVerilog Variable Logic Vector to SPICE Bus	459
Applying Bit-Wise Discipline for SPICE Bus Connections	460
Features Enabled Using the -sv_ms Option	462
Connecting SystemVerilog Hierarchical UDT to Electrical	462
Coercion for SystemVerilog Interfaces, Program, or Checker Blocks	464
Using Wildcard-Named Port Connections in SV Bind Statements	468
Connecting SystemVerilog Interface to DMS/Verilog-AMS (Electrical) Signals	469
Connecting SystemVerilog Multi-Dimensional Array to SPICE/Electrical Ports	472
Build Time & Run Time Performance Improvement for Mixed Signal Designs	474
20	476
Low Power and Mixed-Signal Verification	476
Related Topics	476
Using Common Power Format in Mixed-Signal Designs	477
Power-Smart Connect Modules for CPF	478
Support for Multiple Digital Drivers	480
ie Card Parameters for Power-Smart Connect Modules	482
Power Aware Modeling in Common Power Format	483
Connecting Design Supply Net to CPF Power Domain and Design Net	484
Support for Transition Slope of Power Supply in CPF-Controlled Analog Block	488
Checking Conflicting Power Domains on Mixed-Signal Boundary	489
Using wreal Data Type in Mixed-Signal Designs with CPF	490
Specifying Boundary Ports and Macro Models	493
Using Electrical Expressions in CPF	494
Feedthrough Wire Analysis	498
Using the IEEE 1801 Standard in Mixed-Signal Designs	498
Power Smart Connect Modules for IEEE 1801	502
Specifying Power Intent on Mixed-Signal Blocks	503
Power-Aware Analog/RNM Blocks	505
Specifying Parameter Values for Connect Modules on Power Supply Nets	509
Automatic Supply Connections in Mixed-Signal Designs with Liberty Model	510
Enabling SystemVerilog Interface in LP-MS Designs	528
21	529
SystemC Integration for Mixed-Signal Verification	529
Related Topic	529
Instantiating Verilog-AMS and VHDL-AMS in SystemC	529
Preparing and Using Wrappers and Shells	530

Preparing Interface Modules	531
Guidelines for Using AMS Modules in SystemC Models	533
The xmshell Command	534
22	537
Incremental Elaboration for Mixed-Signal	537
Use Models of Incremental Elaboration	538
Partitioning for UVM Test Modifications	538
Partitioning for a Stable DUT	538
Flexible Partitioning for Complex SOCs	539
Unsupported Scenarios of Incremental Elaboration for Mixed-Signal	539
Limitations of Incremental Elaboration for Mixed-Signal	540
Related Topics	541
Incremental Elaboration at SV-RNM Partition Boundary	542
Auto-Partitioning Designs with SV-RNM Ports	544
Coercion of User-Defined Nettype (UDN) across Partition Boundaries	546
Incremental Elaboration in Designs with UNL Netlists	547
Incremental Elaboration in OOMR Connections	548
Incremental Elaboration in Low-Power Mixed-Signal Designs	550
Cloning of Replicated Modules	553
Appendix A: Updating Legacy Libraries and Netlists	556
Appendix B: Tcl-Based Debugging	558
Related Topics	561
analog	561
analog Command Syntax	561
analog Command Options	561
call	562
call Command Syntax	562
call Command Options	562
call Command Arguments	563
call Command Examples	564
deposit	565
deposit Command Syntax	566
deposit Command Options	567
deposit Command Example	567
describe	569
describe Command Syntax	570

describe Command Options	570
describe Command Examples	570
drivers	572
drivers Command Syntax	573
drivers Command Options	573
drivers Command Examples	573
drivers Command Report Format	578
exit	580
exit Command Syntax	580
exit Command Options	581
finish	581
finish Command Syntax	581
finish Command Options	581
finish Command Examples	582
force	582
force Command Syntax	584
force Command Options	585
force Command Examples	585
get_analog_param	588
get_analog_param Command Syntax	588
get_analog_param Command Options	588
probe	589
probe Command Syntax	589
probe Command Options	591
probe Command Examples	597
Probing Built-In and User-Defined Nettypes	600
Probing SPICE Ports	602
release	603
release Command Syntax	604
release Command Options	604
release Command Examples	604
reset	606
reset Command Syntax	607
reset Command Options	607
reset Command Example	607
restart	607
restart Command Syntax	608

restart Command Options	609
restart Command Limitations	609
restart Command Examples	610
run	610
run Command Syntax	611
run Command Options	611
run Command Examples	612
save	613
save Command Syntax	614
save Command Options	614
save Command Examples	616
scope	621
scope Command Syntax	622
scope Command Options	622
scope Command Example	624
set_analog_param	630
set_analog_param Command Syntax	631
set_analog_param Command Options	631
status	633
status Command Syntax	633
status Command Options	633
status Command Example	633
stop	633
Syntax	634
Modifiers and Options	635
Example	648
Tcl Expressions as Arguments	655
strobe	656
strobe Command Syntax	658
strobe Command Options	658
strobe Command Examples	659
time	664
time Command Syntax	665
time Command Options	665
time Command Examples	665
value	666
value Command Syntax	666

value Command Options	666
value Command Example	668
where	669
where Command Syntax	669
where Command Options	669
where Command Example	670
Specifying Unnamed Branch Objects in Tcl Commands	670
Appendix C: Source Protection	671
Verilog-AMS Protection	671
Protection Guidelines for Automatically Inserted Connect Modules	671
Forced Use of CMI 3.0	672
Related Topics	672
Using xmprotect for Source Protection	672
Examples of Using the Protection Pragmas	674
Protecting All Modules in a Source Description	676
Using spectre_encrypt for Source Protection	677
The spectre_encrypt Command	678
Appendix D: Profiler for Performance Analysis	679
Mixed-Signal Simulation Summary	680
Digital Simulation Profile Results	680
Stream Counts	680
Most Active Modules	681
Stream Type Summary Counts	682
Analog Simulation Profile Results	682
Related Topic	683
Appendix E: Migration of prop.cfg file to an amsd Block	684
string prop sourcefile Translation	684
cell, inst, and path Translations	684
cell	684
inst	685
path	685
Stub Instance Translations	686
default Translations	687
string prop hdl_cell Translation	688
string prop sim_stub Translation	688
string prop sourcefile_opts Translations	689

28	691
Glossary	691
A	691
B	691
C	692
D	692
E	693
G	693
H	693
I	693
M	694
N	694
O	695
P	695
R	696
S	697
T	697
U	698
V	698
General Terms Related to Simulation	698

Product and Licensing Information for Mixed-Signal Designs

Mixed-signal licenses have been updated to support the new Xcelium Limited license in addition to the Xcelium Single Core or Xcelium Multi Core license for digital-centric mixed-signal features. Xcelium Limited includes basic digital features such as SystemVerilog Design and SystemVerilog Assertion support. For the list of features that you can avail using the Xcelium Limited license, see [Table, Feature Comparison of Xcelium Limited and Xcelium Single Core/Multi Core Licenses](#).

And, Spectre AMS Designer has been updated to include the combined features of spectre AMS Connector and Xcelium Limited.

Spectre AMS Designer and Spectre AMS Connector enables you to run AMS features in Xcelium. In addition, a MMSIM token, Spectre MMSIM token with AMS, 90005, is available that enables you to access Spectre AMS Connector and Spectre AMS Designer product licenses. Spectre AMS Connector works with Xcelium only and does not include any analog or digital simulation features. It enables mixed-signal simulation by connecting the Spectre and Xcelium simulators.

Spectre AMS Designer and the Spectre AMS Connector can be used to run mixed-signal simulation features, such as Verilog 1364, VHDL 1076, Verilog-AMS based modeling (including Verilog-AMS Wreal and Verilog like packed wire), and VHDL-AMS.

AMS features, such as Verilog-AMS/VHDL-AMS, can be run in Xcelium using the Spectre AMS Designer license (or 2 MMSIM tokens).

To access digital mixed-signal functionalities, such as SystemVerilog Real Number Modeling (SV-RNM) in Xcelium, you need an Xcelium Mixed-Signal App license in addition to the Spectre AMS Designer License (or 2 MMSIM tokens).

Any other digital-centric mixed-signal features, such as SystemVerilog and mixed-signal, low power and mixed-signal (LP+MS), and so on, require an Xcelium Single Core or Xcelium Multi Core license and an Xcelium Mixed-Signal App license.

Also, when SV-Real is used in covergroup (SV-Real + Coverage), it would require Xcelium Single Core or Xcelium Multi Core license and an Xcelium Mixed-Signal App license.

If an analog solver is also used, then, the Spectre AMS Connector license (or 1 MMSIM token) is also required.

Analog solvers require their own licenses.

Table 1.1: Feature Comparison of Xcelium Limited and Xcelium Single Core/Multi Core Licenses

Features	Xcelium Limited	Xcelium Single Core/Multi Core
Verilog Simulation	✓	✓
VHDL Simulation	✓	✓
Code Coverage	✓	✓
HAL	✓	✓
PSL/IAL	✓	✓
SimVision	✓	✓
Integrated Metric Center	✓	✓
Works with DMS, AMS	✓	✓
Multi-core Code Generation	✓	✓
SVA, SystemVerilog for Design	✓	✓
SystemVerilog Testbench	✗	✓
e/Specman Testbench	✗	✓
SystemC	✗	✓
Low Power Verification	✗	✓
UVM including ML and Accel	✗	✓
Multi-core Waveform Dump	✗	✓
Incremental Elaboration	✗	✓

Related Topics

- [License Checkout Order for Mixed-Signal Designs](#)
- [License Requirement Report](#)
- [Feature-Specific License Checkout Order](#)
- [Platform Support](#)

Platform Support

The XCELIUM 22.09 MAIN and later releases include 64-bit versions of the Spectre AMS Designer simulator. The current release of Xcelium supports the following platforms:

Table 2.1: Xcelium Support Matrix

Architecture (64-bit)	x86_64 Arm LOP Arm and LOP are officially supported for XCELIUMAGILE only.
Compiler	GCC 9.3 (primary) GCC 6.3 (secondary) Important The LOP architecture will continue to use GCC 6.3 and GCC 4.8.3.
OS Version	CentOS 7.4 RHEL 7.4 RHEL 7.6 RHEL 7.9 RHEL 8.1 RHEL 8.3 SLES12/SP3 RHEL 8.x is a 64-bit platform; hence, Xcelium is supported only in 64-bit mode. For more information, read the RHEL8.x support in Xcelium article on COS.
NFS	NFSv3
Xcelium Single-Core	32/64-bit
Xcelium Multi-Core	64-bit only

The current release of Spectre AMS Designer supports the following platforms:

Platform and Architecture (64-bit)	LINUX RHEL 7 LINUX RHEL 8 LINUX SLES 12
Development OS	LINUX RHEL 7

For information on how to run the simulator in 64-bit mode, see [Versions of the Spectre AMS Designer Simulator](#).

Related Topics

- [Product and Licensing Information for Mixed-Signal Designs](#)
- [Installing Xcelium](#)
- [License Checkout Order](#)
- [License Reason Printing](#)
- [Feature-Specific License Checkout Order](#)

License Checkout Order for Mixed-Signal Designs

To run AMS features in Xcelium, you must have access to a corresponding license or combination of licenses. You can change the license checkout order for the Spectre simulators by using the `-spectre_args +lorder` option.

To change the checkout order of Spectre AMS licenses used in the Xcelium simulator, use the `-ams_lorder` option. You can use the following arguments with the `-ams_lorder` option:

- `SAMSD`: Checks out the Spectre AMS Designer license.
- `SAMSC`: Checks out the Spectre AMS Connector license.
- `MMSIM2`: Checks out the MMSIM2 license. In this case, the MMSIM2 license has higher priority than licenses with one MMSIM.
- `MMSIM`: Checks out only the MMSIM license. In this case, licenses with one MMSIM have higher priority than 2MMSIM.
- `XCELIUM`: Check out only the Xcelium-based licensing for digital+mixed-signal simulations, with no analog solver, specify `-ams_lorder XCELIUM`. You cannot use the other four checkout order arguments along with the XCELIUM license order.

To check out the license in the order MMSIM, SAMSD, and SAMSC, specify `-ams_lorder`

`MMSIM:SAMS:D:SAMSC`.

AMS features in Xcelium can be accessed using the product a-la-carte licenses (Spectre AMS Connector or Spectre AMS Designer) or MMSIM tokens. If both are available, product a-la-carte licenses are checked out first.

Additionally, to specify the license checkout order of Xcelium simulator with the mixed-signal option, you can use the `-license_order` command. You can use the following arguments:

- `XCELIUM_SC_ONLY`: Checks out only Xcelium Single Core as the main Xcelium license.

For mixed-signal simulations with advanced MSV and Xcelium centric feature set requiring no analog solver, *xrun* checks out Xcelium Single Core and Xcelium MixedSignal App.

For mixed-signal simulations with advanced MSV and Xcelium centric feature set requiring an analog solver, *xrun* checks out Xcelium Single Core, Xcelium MixedSignal App, Spectre AMSD Connector (SAMS).

- `XCELIUM_SC_DMS`: Checks out only Xcelium Single Core and Xcelium Mixed-Signal App licenses for simulation.

For mixed simulation, *xrun* checks out only Xcelium Single Core and Xcelium Mixed-Signal App, when analog solver is not used. And, *xrun* does not check out SAMSD or 2MMSIM to replace Xcelium Single Core, when analog solver is used.

For pure digital designs without the Mixed-Signal App feature, this option behaves the same as `-license_order XCELIUM_SC_ONLY`.

Related Topics

- [Product and Licensing Information for Mixed-Signal Designs](#)
- [License Reason Printing](#)
- [Feature-Specific License Checkout Order](#)
- [Platform Support](#)

License Requirement Report

You can use the `-licinfo` command-line option with `xrun` command to determine the simulation-time licenses required for a specified design. The `-licinfo` option provides additional information about the features that use a DMSO license, which is checked out for mixed-signal design.

The `-licinfo` option does not check out any licenses and does not run the simulation. The software only loads the snapshot header to identify the current license requirements.

The `-licinfo` option generates a report on the licensing requirements using the following parameters:

- The contents of the design (AMS, SystemC, SystemVerilog, Verilog, VHDL, and so on).
- Other options that are specified on the command line.
- Any arguments specified using the `-uselicense` Option.

Example

The following is an example of the Xcelium Mixed-Signal App license being checked out and it also lists the feature that is using the Xcelium Mixed-Signal App license:

```
1 AMS_Designer_Link AND 1 Xcelium_MixedSignal_App
OR
2 Virtuoso_Multi_mode_Simulation AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Limited_Single_Core AND 1 AMS_Connector AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Limited_Single_Core AND 1 Virtuoso_Multi_mode_Simulation AND 1
Xcelium_MixedSignal_App
OR
1 XceliumLimitedSingleCoreLegacy AND 1 Incisive_HDL_Simulator AND 1 AMS_Connector AND 1
Xcelium_MixedSignal_App
OR
1 XceliumLimitedSingleCoreLegacy AND 1 Incisive_HDL_Simulator AND 1
Virtuoso_Multi_mode_Simulation AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Single_Core AND 1 AMS_Connector AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Single_Core AND 1 Virtuoso_Multi_mode_Simulation AND 1
Xcelium_MixedSignal_App
OR
1 Xcelium_Single_Core_Legacy AND 1 Incisive_Enterprise_Simulator AND 1 AMS_Connector
```

```
AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Single_Core_Legacy AND 1 Incisive_Enterprise_Simulator AND 1
Virtuoso_Multi_mode_Simulation AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Safety_Sim AND 1 AMS_Connector AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Safety_Sim AND 1 Virtuoso_Multi_mode_Simulation AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Safety AND 1 AMS_Connector AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Safety AND 1 Virtuoso_Multi_mode_Simulation AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Multi_Core AND 1 AMS_Connector AND 1 Xcelium_MixedSignal_App
OR
1 Xcelium_Multi_Core AND 1 Virtuoso_Multi_mode_Simulation AND 1 Xcelium_MixedSignal_App
Loading snapshot worklib.top:vams ..... Done
```

The Xcelium Mixed Signal App License is needed because the following DMS technologies are used in the design:

- * SystemVerilog Real Number Modeling (SV-RNM) Technology
- * Advanced Mixed Signal Algorithms/Technology (Functionality, Performance or Debug)

Debugging features such as Tcl-commands `force/release on wreal` and `save/restart` require Xcelium Mixed-Signal App license; however, this cannot be determined by `-licinfo`.

Related Topics

- [Product and Licensing Information for Mixed-Signal Designs](#)
- [Feature-Specific License Checkout Order](#)
- [Platform Support](#)

Feature-Specific License Checkout Order

The following table lists the default license checkout order for different mixed-signal features.

Feature	Default License Checkout Order

Verilog-AMS, or Verilog, or VHDL or VHDL-AMS design	• Spectre AMS Designer • 2 MMSIM tokens • Spectre AMS Connector + Xcelium Single Core • 1 MMSIM token + Xcelium Single Core • Spectre AMS Connector + Xcelium Multi Core • 1 MMSIM token + Xcelium Multi Core
Verilog-AMS, or Verilog, or VHDL or VHDL-AMS design + WREAL	• Spectre AMS Designer • 2 MMSIM tokens • Spectre AMS Connector + Xcelium Limited • 1xMMSIM + Xcelium Limited • Spectre AMS Connector + Xcelium Single Core • 1 MMSIM token + Xcelium Single Core • Spectre AMS Connector + Xcelium Multi Core • 1 MMSIM token + Xcelium Multi Core

Verilog-AMS, or Verilog, or VHDL, or VHDL-AMS design + SVRNM	• Spectre AMS Designer + Xcelium Mixed-Signal App • 2 MMSIM tokens + Xcelium Mixed-Signal App • Spectre AMS Connector + Xcelium Limited +Xcelium Mixed-Signal App • 1xMMSIM + Xcelium Limited + Xcelium Mixed-Signal App • Spectre AMS Connector + Xcelium Single Core + Xcelium Mixed-Signal App • 1 MMSIM token + Xcelium Single Core + Xcelium Mixed-Signal App • Spectre AMS Connector + Xcelium Multi Core + Xcelium Mixed-Signal App • 1 MMSIM token + Xcelium Multi Core + Xcelium Mixed-Signal App
Verilog-AMS, or Verilog, or VHDL, or VHDL-AMS design + SVRNM + SVA	• Spectre AMS Designer + Xcelium Mixed-Signal App • 2xMMSIM + Xcelium Mixed-Signal App • Spectre AMS Connector + Xcelium Limited +Xcelium Mixed-Signal App • 1xMMSIM + Xcelium Limited + Xcelium Mixed-Signal App • Spectre AMS Connector + Xcelium Single Core + Xcelium Mixed-Signal App • 1xMMSIM + Xcelium Single Core+ Xcelium Mixed-Signal App • Spectre AMS Connector + Xcelium Multi Core + Xcelium Mixed-Signal App • 1xMMSIM + Xcelium Multi Core + Xcelium Mixed-Signal App

Verilog-AMS, or Verilog, or VHDL, or VHDL-AMS design + UVM/SVTB

- Spectre AMS Connector + Xcelium Single Core + Xcelium Mixed-Signal App
- 1 MMSIM token + Xcelium Single Core + Xcelium Mixed-Signal App
- Spectre AMS Connector + Xcelium Multi Core + Xcelium Mixed-Signal App
- 1 MMSIM token + Xcelium Multi Core + Xcelium Mixed-Signal App

Verilog-AMS or Verilog, or VHDL or VHDL-AMS design +UVM/SVTB + SVRNM	• Spectre AMS Connector+ Xcelium Single Core + Xcelium Mixed-Signal App • 1 MMSIM token + Xcelium Single Core + Xcelium Mixed-Signal App • Spectre AMS Connector + Xcelium Multi Core + Xcelium Mixed-Signal App • 1 MMSIM token + Xcelium Multi Core+ Xcelium Mixed-Signal App
Digital (Verilog/VHDL) + SVRNM	• Xcelium Limited + Xcelium Mixed-Signal App • Xcelium Single Core + Xcelium Mixed-Signal App • Spectre AMS Designer + Xcelium Mixed-Signal App • 2 MMSIM tokens + Xcelium Mixed-Signal App • Xcelium Multi Core + Xcelium Mixed-Signal App
Digital (SystemVerilog) + SVRNM	• Xcelium Limited + Xcelium Mixed-Signal App • Xcelium Single Core + Xcelium Mixed-Signal App • Xcelium Multi Core + Xcelium Mixed-Signal App

Digital (Verilog/VHDL, SystemVerilog Design or SVA) + WREAL	• Spectre AMS Designer • 2 MMSIM tokens • Xcelium Limited + Xcelium Mixed-Signal App • Xcelium Limited + Spectre AMS Connector • Xcelium Limited + 1xMMSIM • Xcelium Single Core + Xcelium Mixed-Signal App • Xcelium Single Core + Spectre AMS Connector • Xcelium Single Core + 1 MMSIM token • Xcelium Multi Core + Xcelium Mixed-Signal App • Xcelium Multi Core + Spectre AMS Connector • Xcelium Multi Core + 1 MMSIM token
Digital (SystemVerilog Testbench)+ WREAL	• Xcelium Single Core + Xcelium Mixed-Signal App • Xcelium Multi Core + Xcelium Mixed-Signal App

Related Topics

- [Product and Licensing Information for Mixed-Signal Designs](#)
- [Feature-Specific License Checkout Order](#)
- [License Requirement Report](#)
- [Platform Support](#)

Typographic and Syntax Conventions

Special typographical conventions are used to distinguish certain kinds of text in this document. The formal syntax used in this reference uses the definition operator, `::=`, to define the more complex elements of the Verilog-AMS language in terms of less complex elements.

- Lowercase words represent syntactic categories. For example,

`module_declaration`

Some names begin with a part that indicates how the name is used. For example,

`node_identifier`

represents an identifier that is used to declare or reference a node.

- Boldface words represent elements of the syntax that must be used exactly as presented (except as noted below). Such items include keywords, operators, and punctuation marks. For example,

endmodule

- The shortest permitted abbreviation is shown by capital letters but you can use either upper or lowercase letters in your code. For example, the syntax

-Checktasks

means that you can type the option as `-checktasks`, `-CHECKTASKS`, `-ch`, `-CH`, `-cH`, and so on.

- Vertical bars indicate alternatives. You can choose to use any one of the items separated by the bars. For example,

`attribute ::=`

```
    abstol
|  access
|  ddt_nature
|  idt_nature
|  units
```

- | **huge**
- | **blowup**
- | identifier

- Square brackets enclose optional items. For example,

input declaration ::=

```
input [ range ] list_of_port_identifiers ;
```

- Braces enclose an item that can be repeated zero or more times. For example,

list_of_ports ::=

```
( port { , port } )
```

- Code examples are displayed in constant-width font.

```
/* This is an example of the font used for code.*/
```

- Variables are in italic font, like this: *allowed_errors*.

- Keywords, filenames, names of natures, and names of disciplines are set in constant-width font, like this: keyword, file_name, name_of_nature, name_of_discipline.

- If a statement is too long to fit on one line, the remainder of the statement is indented on the next line, like this:

```
qgf = width*length*cfbb*(vgfs - wkf - qb/(2*cbb) -
      (vgbs - vfbb + qb/(2*cob))) + qgf_par ;
```

Spectre AMS Designer Simulator and Xcelium Simulator with Mixed-Signal Option Capabilities

This topic talks about the AMS Designer features that can be run using the Spectre AMS Designer (AMS Designer) simulator and the Xcelium simulator with mixed-signal option. It includes the features related to Verilog-AMS, and VHDL-AMS along with SPICE or Verilog-A, and related use models.

The Spectre AMS Designer simulator covers features related to Verilog-AMS, and VHDL-AMS along with SPICE or Verilog-A, and related use models.

The Xcelium mixed-signal simulator covers features related to Verilog-AMS, VHDL-AMS, SystemVerilog Real Number Modeling (SVRNM), SystemVerilog (SV) and mixed-signal features along with SPICE or Verilog-A, and other digital-centric mixed-signal technologies, such as low power + MS, code coverage + MS, and so on.

The features covered under the Spectre AMS Designer Simulator can also be run using the Xcelium simulator with mixed-signal option.

The two mixed-signal use models, AMS Designer Virtuoso use model (AVUM) and AMS Designer Xcelium use model (AXUM), are common to both Spectre AMS Designer and Xcelium mixed-signal simulators.

The AVUM use model is run in Virtuoso® ADE Explorer, which provides seamless integration of the Spectre AMS Designer and Xcelium Mixed-Signal simulator. Integrating the AMS Designer simulator with ADE Explorer creates a design environment with the look and feel expected by the analog and mixed-signal designers who already use ADE Explorer. When using this integration, you can access designs using the same tools you currently use for pure analog and mixed-signal designs.

The topics are organized based on the features that are common to both Spectre AMS Designer and Xcelium mixed-signal simulators and the features that are applicable only to the Xcelium mixed-signal simulator.

Related Topics

- [AMS Designer Simulator](#)

- [XRUN Invocation for Mixed-Signal Designs](#)
- [Using xrun for AMS Simulation](#)
- [Using SimVision with the AMS Simulator](#)
- [MSV Product Homepage](#)
- IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS Changes), IEEE Std 1076.1. Available from IEEE.
- Instance-Based View Switching Application Note

AMS Designer Simulator

The AMS Designer simulator is a mixed-signal simulator that supports the Xcelium™ digital engine and the analog solvers, Spectre, APS, XPS-MS, and Spectre X. The Spectre solver provides a proven, highly accurate simulation and the APS solver provides the fastest simulation speed for designs with more than 50K elements in SPICE blocks.

See the following topics for more information:

Related Topics

- [Language Support](#)
- [Running the AMS Designer Simulator](#)
- [Analog Solver Information](#)
- [Analog Solvers Supported by AMS Designer Simulator](#)
- [Advanced Analog Solver Features](#)
- [Compatible Spectre Version Control File](#)
- [The spectre_compatible_version.xml File Location](#)

Language Support

Except as noted, the Spectre® AMS Designer simulator complies with

- The IEEE 1364 standard described in *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language* (IEEE Std 1364-2005), published by the IEEE
- VHDL-AMS (IEEE 1076.1-1999)
- The OVI 2.0 description of the language described in *OVI Verilog Hardware Description Language Reference Manual*, Version 2.0, published by Open Verilog International
- The Verilog®-XL implementation of the Verilog language described in the *Verilog-XL Reference*
- The Verilog®-AMS language standard described in the *Verilog-AMS Language Reference Manual*, Version 2.2, published by Accellera
- The SystemVerilog language standard described in the [SystemVerilog Reference](#)

You can use the `-ieee1364` command-line option when you run the `xmvlog` compiler and the `xmelab` elaborator to check your code for compatibility with the IEEE standard.

Related Topics

- [Unsupported Elements of Verilog-AMS](#)
- [Cadence Verilog-AMS Language Reference](#)

Versions of the Spectre AMS Designer Simulator

The Spectre AMS Designer simulator supports 64-bit operations on all platforms that are officially supported by Cadence. You must run all software in one mode or the other; you cannot mix modes. For example, if you run `xmvlog` in 64-bit mode, you must also run the 64-bit versions of `xmelab` and `xmsim`.

To run the Spectre AMS Designer simulator in 64-bit mode, do one of the following:

- Set the path variable and the library path environment variable to point to the executable files that enable the software to run in 64-bit mode.
- Use the `-64bit` command-line option when you run each executable (`xmvlog`, `xmelab`, `xmsim`, or `xrun`).

When you use the `-64bit` command-line option, the software sets the `PATH` variable and the library path environment variable to run the software in 64-bit mode. Do not use this command-line option when linking the software to 64-bit applications, such as PLI, VPI, or VHPI. Set the path variable and the library path environment variable, instead.

- Set the `CDS_AUTO_64BIT` environment variable.

✔ Use the `xmbits` command, as shown below, to see the bit mode that has been set up to run the software:

```
xmbits
64
```

The `-version` command-line option also reports the version of the simulator being used. For example:

```
xrun -version
TOOL: xrun(64) 22.03-s001
```

For additional information, see *Running the Simulator in 64-Bit Mode* in the *Overview of Running the Xcelium Simulator* book.

Setting the Path Variable to Point to the 64-Bit Version

You can find the 64-bit executables installed at `install_dir/tools/bin/64bit`.

To set the path variable to point to the 64-bit executables:

- Use one of the following commands (depending on the shell that you are running).

```
setenv CDS_AUTO_64BIT ALL
setenv PATH /xceliumInstallDir/tools/bin:$PATH
setenv PATH /spectreInstallDir/tools.lnx86/bin:$PATH
```

Library Path Environment Variable

You need to set the `LD_LIBRARY_PATH` environment variable before you can use the three-step method to run the simulation. However, setting this path is not required if you are using the `xrun` command to run the simulation.

- For non-SUSE Linux, set the library path environment variable, as follows:

```
setenv LD_LIBRARY_PATH install_dir/tools/lib/64bit:install_dir/tools/lib:${LD_LIBRARY_PATH}
```

- For SUSE Linux, set the library path environment variable, as follows:

```
setenv LD_LIBRARY_PATH
install_dir/tools/lib/64bit/SuSE:install_dir/tools/lib/64bit:install_dir/tools/lib/SuSE:install_dir/tools/lib:${LD_LIBRARY_PATH}
```

Running the AMS Designer Simulator

There are two ways to run the AMS Designer simulator:

- Single-step: In this approach, you issue one command, the `xrun` command. This command automatically runs the three steps in turn:

the `xmvlog` compiler, the `xmelab` elaborator, and the `xmsim` simulator. Cadence recommends this easier-to-use use model.

Setting of the library path environment variable `LD_LIBRARY_PATH` is not required if you are using the `xrun` command to run the simulation.

- Three-step: In this approach, you run `xmvlog`, `xmelab`, and `xmsim` separately.

Related Topic

- [Single-Step Approach to Running a Simulator](#)

Single-step Approach to Running a Simulator

The `xrun` command lets you run the simulator by specifying all input files and command-line options on a single command line. `xrun` takes files from different simulation languages, such as Verilog, SystemVerilog (SV), VHDL, Verilog-AMS, VHDL-AMS, Specman `e`, as well as C and C++ general programming language files, and compiles them using the appropriate compilers based on their file extensions. After compiling the input files, `xrun` automatically runs `xmelab` to elaborate the design and then `xmsim` to simulate the design.

Related Topics

- [XRUN Invocation for Mixed-Signal Designs](#)
- [Using the AMS Designer Simulator for Design Verification](#)

AMS Designer Flex Mode

AMS designer flex mode gives you the flexibility to choose independent versions of Xcelium and Spectre simulation engines. It also provides you direct access to enhancements in the Spectre engine. Starting XCELIUM 20.03 s005, AMS designer flex mode is enabled by default if SPECTRE 19.1 ISR6 or later is found in the system path or is specified using the `xrun -Spectre_path` option.

From the XCELIUM 20.09 release, you need SPECTRE 19.1ISR6 or later release in your path when running AMS (Analog Mixed Signal) or RNM (Real Number Modeling) applications that include Spectre files (*.scs)

```
<SPECTRE_install_dir>/bin
```

Analog Solver Information

Use the following `xmsim` commands to display the exact SPECTRE ISR version number, without having to run a simulation.

```
xmsim -spectre_args -W
```

```
xmsim -spectre_args -V
```

The SPECTRE version printed out by the above commands applies to all SPECTRE products of Spectre and APS that are integrated with AMS Designer.

In addition, the `xmsim` simulation log contains the SPECTRE ISR version number information at the beginning of the log. This version message is exactly the same as the one that is shown in the simulation logs of SPECTRE products in SPECTRE releases.

```
Cadence (R) Virtuoso (R) Spectre (R) Circuit Simulator
```

```
Version 19.1.0.373 64bit -- 15 May 2020
```

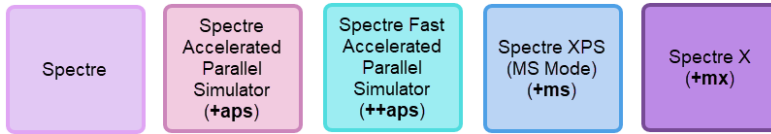
When running in AMS Designer in flex mode the Spectre installation used is printed in the log. For example:

```
AMSD Flexible Release Matrix --- using Spectre from installation:
```

```
/<SPECTRE_install_dir>/tools.lnx86/spectre/bin
```

Analog Solvers Supported by AMS Designer Simulator

The figure below shows the analog solvers that are supported with the AMS Designer simulator:



The following table shows how to enable the analog solvers in the AMS Designer simulator:

Analog Solver	How to enable...
Spectre (default)	<code>xrun amscf.scs</code>
APS	<code>xrun -spectre_args "+aps" amscf.scs</code>
Fast APS	<code>xrun -spectre_args "++aps" amscf.scs</code>
Spectre X	<code>xrun -spectre_args "+preset=cx ax mx lx vx" amscf.scs</code>
XPS (MS mode) By default, APS is enabled for the analog partition.	<code>xrun -spectre_args "+preset=cx ax mx lx vx" amscf.scs</code>
XPS (MS mode) With fast APS enabled for the analog partition.	<code>xrun -spectre_args "+ms +preset=cx ax mx lx vx" amscf.scs</code>

Related Topics

- [AMS Spectre XPS MS Mode](#)
- [AMS Designer with Spectre FX](#)

Advanced Analog Solver Features

You can access the following advanced feature of Spectre GXL with the AMS Designer simulator.

Spectre GXL Feature	AMS Designer + Spectre Solver
Parasitic reduction	See Turning On Spectre Parasitic Reduction

Compatible Spectre Version Control File

You can view the compatible Spectre release version for a given Xcelium release using the `spectre_compatible_version.xml` file, located at:

`tools.lnx86/affirma_ams/etc/files/spectre_compatible_version.xml`

It is recommended to not modify this file.

To modify the compatible Spectre release version, you can create and edit a new `spectre_compatible_version.xml` file in one of the following locations:

- `$CWD/.amsd`
- `$CDS_WORKAREA/.amsd`

Example

```
<?xml version="1.0" encoding="UTF-8"?>
<spectre>
<allow-newer version="19.1.0.348.isr6"/>
<allow version="19.1.0.348.isr6">
</spectre>
```

This sets the compatible Spectre release version to Spectre 19.1.0.348.isr6 and higher versions.

The following table describes the options you can use in the newly created `spectre_compatible_version.xml` for editing the compatible Spectre release version for the current Xcelium release.

Option	Description
<code>allow-newer version</code>	Sets the compatible Spectre release version to any version newer than the one specified through this option. Example: <pre><allow-newer version="19.1.0.348.isr6"></pre> Sets the compatible Spectre version to any version above 19.1.0.348.isr6.
<code>allow version</code>	Specifies the compatible Spectre release version for the current Xcelium release. If set to <code>validated</code>, the compatible release version is set to the version validated with the current Xcelium release. Example: <pre><allow version="validated"/></pre> Sets the Spectre compatible version to the version validated with the current Xcelium release.

Related Topics

[The spectre_compatible_version.xml File Location](#)

The spectre_compatible_version.xml File Location

To find the `.amsd/spectre_compatible_version.xml` file, use the following search order.

<code>\$CWD</code>
<code>\$CDS_WORKAREA</code> (user work area, if defined)
<code>\$CDS_SEARCHDIR</code> (if defined)
<code>\$HOME</code>
<code>\$CDS_PROJECT</code> (project area, if defined)
<code>\$CDS_SITE</code> (site setup, if defined)
<code>xcelium_install_dir/share</code>

Related Topics

[Compatible Spectre Version Control File](#)

Analog Simulation Control File

The analog simulation control file is an ASCII text file containing commands that control the behavior of the analog solvers. It must not contain any subcircuit or model definitions or other Spectre/SPICE commands. The analog simulation control file must have `.scs` as the file extension in order to establish the Spectre language as the default language for the file. Commands in the analog simulation control file apply to `xmsim` (or to the simulation phase of `xrun`).

Related Topics

- [Switching Languages in the Analog Simulation Control File](#)
- [Spectre Language Statements in the Analog Simulation Control File](#)
- [Passing the Analog Simulation Control File to xrun](#)
- [Specifying Controls for the Analog Solvers](#)

Switching Languages in the Analog Simulation Control File

Although Cadence recommends that you put all your analog control file statements in the Spectre language section of the analog simulation control file, you can switch to the SPICE language.

To do this:

- Specify the following statement at the point in the file:

```
simulator lang=spice
```

You cannot use Spectre language statements in the SPICE language section of the control file.

Using Spectre lookup Rules

You can use Spectre lookup rules so that the simulator is sensitive to the casing of node and hierarchical names but not to the casing of keywords, parameters, and units.

To do this:

- Specify the `lookup` option on the `simulator lang` line.

```
simulator lang=spice lookup=spectre
```

You must specify `lookup=spectre` when you use `.meas`, `.probe`, and `.print` SPICE commands. Ensure that use the full hierarchical node name in these commands.

At any point where you want to change back to the Spectre language, type the following statement in the file:

```
simulator lang=spectre
```

Example

```
simulator lang=spice lookup=spectre  
.meas myMeasure max v(myTop.myNode)
```

Specifies the `.meas` statement in the analog simulation control file for a top-level module `myTop` containing the node `myNode`.

Related Topics

- [Spectre Language Statements in the Analog Simulation Control File](#)
- [Passing the Analog Simulation Control File to xrun](#)
- [Passing the Analog Simulation Control File to xmsim](#)

Spectre Language Statements in the Analog Simulation Control File

The Spectre language is the default language for the file. You must always make the first line of the file as a comment, with slashes at the beginning. Because the simulator uses Spectre lookup rules for both kinds of control statements, it is sensitive to the case of subcircuit, instance, port, node, and vector names.

Stand-alone Spectre always assumes the first line of the top-level netlist to be a comment line, whereas AMS does not do so. Therefore, you must explicitly mark the first line of the SPICE file as a comment. Otherwise, it will result in an error.

For example, consider the following analog simulation control file contents. Notice that the first line is a comment, with slashes at the beginning. You might use the first line to label your control file. The Spectre-language statements that follow specify the data format and the stop time to use.

```
//Control File 3c  
saveNodes options rawfmt=sst2  
timeDom tran stop=5n
```

Related Topics

- [Controls for the Analog Solvers](#)
- [Switching Languages in the Analog Simulation Control File](#)
- [Passing the Analog Simulation Control File to xrun](#)
- [Passing the Analog Simulation Control File to xmsim](#)

Passing the Analog Simulation Control File to xrun

You can specify the analog control file directly on the `xrun` command line.

To pass the analog simulation control file to `xrun`:

- Set the following command.

```
xrun analogControlFileName fileName
```

Example

```
xrun adc.scs sar6bit.v
```

Passes an analog control file called `adc.scs` to `xrun` to compile, elaborate, and simulate a design contained in a file called `sar6bit.v`.

Related Topic

- [Passing the Analog Simulation Control File to xmsim](#)

Passing the Analog Simulation Control File to xmsim

To pass the analog simulation control file to `xmsim`:

- Use the `-analogcontrol` command-line option.

```
xmsim -analogcontrol analogControlFileName ...
```

Example

```
xmsim -analogcontrol adc.scs sar6bit.v
```

Passes an analog control file called `adc.scs` to `xmsim` to compile, elaborate, and simulate a design contained in a file called `sar6bit.v`.

Related Topic

- [Passing the Analog Simulation Control File to xrun](#)

Controls for the Analog Solvers

In the Spectre® AMS Designer simulator, the primary way of controlling the analog solvers is to define an analog simulation control file. The analog simulation control file is an ASCII text file containing commands that control the behavior of the analog solvers. It must not contain any subcircuit or model definitions or other Spectre/SPICE commands. The analog simulation control file must have `.scs` as the file extension in order to establish the Spectre language as the default language for the file. Commands in the analog simulation control file apply to `xmsim` (or to the simulation phase of `xrun`).

The `simulator lang` statement specifies the language mode you want the software to use to evaluate succeeding statements.

Related Topics

- [Analog Simulation Control File](#)
- [Switching Languages in the Analog Simulation Control File](#)

Spectre AMS Designer Analysis Support

Spectre AMS Designer supports the following analyses:

- ac
- alter
- altergroup
- check
- checklimit
- dc
- dspf_options
- emir
- if
- info
- montecarlo
- noise
- options
- reliability
- stb
- sweep
- tran
- xf

Immediate Set Options (options)

The options statement sets or changes program control options. These options take effect immediately and are set while the circuit is read.

```
Name options parameter=value { parameter = value }
```

For more information, see "Immediate Set Options (options)" in the "Analysis Statements" chapter of *Spectre Circuit Simulator Reference*.

The effective value of each parameter of each option is determined by examining the analog simulation control file and the model files.

- If an `options` parameter is specified in the analog simulation control file, the last specification of the value is the effective value. Any corresponding specification in the model file is ignored.

Spectre AMS Designer and Xcelium Simulator Mixed-Signal User Guide

Controls for the Analog Solvers

- If an `options` parameter is not specified in the analog simulation control file but is specified in the model file, the last specification is the effective value.
- Option specifications inside subcircuit definitions take precedence over global specifications.

The parameters and values that you can use with the `options` statement are listed in the following table. Values listed in the parameter syntax are the defaults.

The Spectre and APS solvers support the following parameters.

Parameter	Definition
<code>approx=no</code>	Uses approximate models. Difference between approximate and exact models is generally very small. Values: <code>no</code> , <code>yes</code>
<code>audit=detailed</code>	Prints the time required by various parts of the simulator. Values: <code>no</code> , <code>brief</code> , <code>detailed</code> , <code>full</code>
<code>checklimit_skip_ie_connect_devices</code> <code><0 1 2></code>	Skips the IE elements during device checking for the specified hierarchy level. • 0: No IE element is skipped. • 1: The instances connected directly to IE will be skipped. • 2: The instances connected directly to IE and hierarchy level < 2 will be skipped.
<code>compatible=spectre</code>	Encourages device equations to be compatible with a foreign simulator. This option does not affect input syntax. Values: <code>spectre</code> , <code>spice2</code> , <code>spice3</code> , <code>cdsspace</code> , <code>hspice</code> , <code>spiceplus</code>
<code>currents=selected</code>	Terminal currents to output. Values: <code>all</code> , <code>nonlinear</code> , <code>selected</code>
<code>debug=no</code>	Give debugging messages. Values: <code>no</code> , <code>yes</code>
<code>diagnose=no</code>	Print additional information that might help diagnose accuracy and convergence problems. Values: <code>no</code> , <code>yes</code>
<code>digits</code>	Number of digits used when printing numbers.
<code>error=yes</code>	Give error messages. Values: <code>no</code> , <code>yes</code>
<code>gmin=1e-12 S</code>	Minimum conductance across each nonlinear device.
<code>gmin_check=max_v_only</code>	Specifies that effect of <code>gmin</code> should be reported if significant. Values: <code>no</code> , <code>max_v_only</code> , <code>max_only</code> , <code>all</code>
<code>highvoltage=no</code>	Enables optimized Spectre settings for high voltage designs including voltage, and current binning, excluding VerilogA and dangling nodes from convergence checks, and optimized large capacitance handling. Possible values are <code>no</code> and <code>yes</code> .
<code>homotopy=all</code>	Method used when no convergence on initial attempt of DC analysis. Values: <code>none</code> , <code>gmin</code> , <code>source</code> , <code>dptran</code> , <code>ptran</code> , <code>all</code>
<code>iabstol=1e-12 A</code>	Current absolute tolerance convergence criterion.
<code>ignshorts=no</code>	Silently ignore shorted components. Values: <code>no</code> , <code>yes</code>
<code>info=yes</code>	Give informational messages. Values: <code>no</code> , <code>yes</code>
<code>inventory=detailed</code>	Print summary of components used. Values: <code>no</code> , <code>brief</code> , <code>detailed</code>
<code>limit=dev</code>	Limiting algorithms to aid DC convergence. Values: <code>delta</code> , <code>log</code> , <code>dev</code>
<code>macromodels</code>	Circuit contains macromodels. Providing this information sometimes helps performance.
<code>maxnotes=5</code>	Maximum number of times any notice will be issued per analysis.
<code>maxnotestologfile</code>	Maximum number of times any notice will be printed to the log file per analysis.
<code>maxrsd=0</code>	Threshold below which parasitic node reduction occurs.
<code>maxwarns=5</code>	Maximum number of times any warning message is issued per analysis.

Spectre AMS Designer and Xcelium Simulator Mixed-Signal User Guide

Controls for the Analog Solvers

maxwarnstologfile	Maximum number of times any warning message is printed to the log file per analysis.
minnr=0.0	Threshold below which resistance inside devices is ignored.
mos_method=s	Method used to evaluate BSIM3V3 and BSIM4 models. Values: <code>a</code> (for accelerated, using table models when available), <code>s</code> (for standard, using standard analytical evaluation) If this option is set to <code>a</code> and the corresponding option on a BSIM model card is set to <code>a</code> (the default), the simulator uses table models to simulate the bsim model. If this option is set to <code>a</code> but the corresponding option on a BSIM model card is set to <code>s</code>, the simulator uses standard analytical evaluation for that bsim model. For more information, see the "Analyses" chapter in <i>Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide</i>.
mos_vres=0.05	Voltage increment for the MOSFET table model interpolation grid.
narrate=yes	Narrate the simulation. Values: <code>no</code> , <code>yes</code>
notation	Notation to be used when printing real numbers to the screen.
note=yes	Give notice messages. Values: <code>no</code> , <code>yes</code>
opptcheck=yes	Check operating point parameters against soft limits. Values: <code>no</code> , <code>yes</code>
paramrangefile	Parameter range file. There is no default; if not provided, the AMS Designer simulator does not do any range checking.
pivabs=0	Absolute pivot threshold.
pivotdc=no	Use numeric pivoting on every iteration of DC analysis. Values: <code>no</code> , <code>yes</code>
pivrel=0.001	Relative pivot threshold.
psfversion	Version of the <code>psfxl</code> waveform format. Possible values: <code>1.0</code> and <code>1.1</code>. The waveform files generated by the <code>psfxl</code> 1.0 format (default) are not compressed and could be bigger in size as compared the <code>sst2</code> format. Use the <code>psfxl</code> 1.1 format to generate waveform files that are smaller in size. The 1.1 version is a lossless compression. The new <code>psfxl</code> format is available in the SPECTRE16.1 ISR14 or later releases, in the different modes (classic, <code>+aps</code>, <code>++aps</code>, <code>+ms</code> & <code>+xps</code>). To view the waveform files, you need IC6.1.7 ISR17 or a later release to plot the results with VIVA or from ADE. This new format is supported in Simvision starting Xcelium 17.10.
quantities=no	Print quantities. Values: <code>no</code> , <code>min</code> , <code>full</code>
rawfile="%C:r.raw"	Output raw data file name, optionally including an absolute or relative path.
rawfmt=sst2	Output raw data file format. Values: <code>psfbin</code>, <code>psfascii</code>, <code>fsdb</code>, <code>wdf</code>, <code>sst2</code>, <code>psfxl</code> The specified format affects only analog signals. For AC analysis, the only supported values are <code>psfbin</code> and <code>psfascii</code>. Only signals that you select by using Tcl commands prior to running the AC analysis are saved. <code>fsdb</code> (Fast Signal Database) and <code>wdf</code> formats are supported for transient analysis only. For a transient analysis written to a unified database that can hold both analog and digital signals (which can be created only by using Tcl commands), the only supported value is <code>sst2</code>. For a transient analysis in which analog signals are saved by using either the analog simulation control file or a Tcl file, the supported values are <code>psfbin</code>, <code>psfascii</code>, <code>fsdb</code>, <code>wdf</code>, <code>sst2</code>, and <code>psfxl</code>.
redundant_currents=no	If yes, save both currents through two terminal devices. Values: <code>no</code> , <code>yes</code>
reltol=0.001	Relative convergence criterion.

<code>rforce=1</code>	Resistance used when forcing nodesets and node-based initial conditions.
<code>save=selected</code>	Signals to output. Values: <code>allpub</code> , <code>lvlpub</code> , <code>selected</code> , <code>none</code> , and <code>nooutput</code> . You can set this parameter for the Spectre and APS solvers only.
<code>scale=1</code>	Device instance scaling factor.
<code>scalem=1</code>	Model scaling factor.
<code>sensfileonly=no</code>	Enable or disable raw output of sensitivity results. Values: <code>no</code> , <code>yes</code>
<code>speed</code>	Establishes the tradeoff between performance speed and accuracy. Higher numbers generally result in faster performance but lower accuracy. Values: 1, 2, 3, 4, 5, 6. You can set this parameter for the Spectre and APS solvers only.
<code>temp=27 C</code>	Temperature.
<code>tempeffects=all</code>	Temperature effect selector. Values: <code>vt</code> , <code>tc</code> , <code>all</code>
<code>tnom=27 C</code>	Default component parameter measurement temperature.
<code>topcheck=full</code>	Check circuit topology for errors. Values: <code>no</code> , <code>min</code> , <code>full</code>
<code>useprobes=no</code>	Use current probes when measuring terminal currents. Values: <code>no</code> , <code>yes</code> The following devices always use probes to save currents (even with <code>useprobes=no</code>): <code>port</code> , <code>delay</code> , <code>switch</code> , <code>hbt</code> , <code>transformer</code> , <code>core</code> , <code>winding</code> , <code>fourier</code> , <code>d2a</code> , <code>a2d</code> , <code>a2ao</code> , <code>a2ai</code> . You cannot use this parameter to save currents during an AC analysis.
<code>vabstol=1e-06 V</code>	Voltage absolute tolerance convergence criterion.
<code>warn=yes</code>	Give warning messages. Values: <code>no</code> , <code>yes</code>

Related Topics

- [Comparing the Behavior of save Parameter Specified in an Analysis and the Global save Option](#)
- [Spectre AMS Designer Analysis Support](#)

AC Analysis Statement (ac)

Only the Spectre solver supports AC analysis. The APS solver also does not support AC analysis.

Using the Spectre solver, the AC analysis statement linearizes the circuit about the DC operating point and computes the response to a given small sinusoidal stimulus. You can specify, at most, one AC analysis in your analog simulation control file.

```
Name ac{ parameter=value }
```

When you run an AC analysis using the AMS Designer simulator with the Spectre solver, the following differences apply:

- You can sweep only frequency during an AC analysis.
- You cannot sweep temperatures, component instance parameters (`dev`), component model parameters (`mod`), or netlist parameters (`param`).

When the AMS Designer simulator runs standalone, it writes the results of the AC analysis to a parameter storage format (PSF) file. By default, the software stores the PSF file in a directory called `ascf.raw` (where `ascf` is the name of the analog simulation control file). You can use the `rawfile` parameter of the `options` statement in the analog simulation control file to specify the location of the file.

You use the Tcl `probe` command to specify probes you want to run during the AC analysis.

You can specify the following parameters and values with the `ac` statement. Values listed in the parameter syntax are the defaults.

Parameter	Definition
annotate=sweep	Degree of annotation. Values: no, title, sweep, status, steps
center	Center of sweep.
dec	Points per decade.
lin=50	Number of steps, linear sweep.
log=50	Number of steps, log sweep.
oppoint=no	Determines whether operating point information is computed, and, if so, specifies where the information is sent. Values: no, screen, logfile, rawfile
prevoppoint=no	Uses the operating point computed by the previous analysis. Values: no, yes If a transient analysis precedes the AC analysis, yes causes the AC analysis to use the final operating point computed for the transient analysis. no causes the AC analysis to use an operating point that is computed independently of any previously computed operating point.
readns	File that contains estimate of DC solution (nodeset).
save	Signals to output. Values: all, lvl, allpub, lvlpub, selected, none This parameter is ignored. For AC analysis, the signals to be saved must be specified by using a <code>probe</code> command in the Tcl input file.
span=0	Sweep limit span.
start=0	Start sweep limit.
stats=no	Analysis statistics. Values: no, yes
step	Step size, linear sweep.
stop	Stop sweep limit.
title	Analysis title.
values=[...]	Array of sweep values.

Related Topics

- [Mixed-Signal DC Initialization](#)

Noise Analysis (noise)

Noise analysis linearizes the circuit about the operating point and computes the noise spectral density at the output. You can enable noise analysis in AMS designer by setting the `AMS_ENABLE_NOISE` environment variable to `YES`. You can specify the noise analysis statement in your analog simulation control file as follows:

```
Name [p] [n] noise parameter=value ...
```

The optional terminals (p) and (n) specify the output of the circuit. If you do not specify the terminals, you must specify the output with a probe component.

For information on noise analysis parameters, refer to the Noise Analysis section in the *Spectre Circuit Simulator Reference* manual.

For small signal noise analysis, the AMS designer simulator supports the following noise sources specified in the Verilog-AMS or Verilog-A files:

- `white_noise`
- `flicker_noise`
- `noise_table`

Related Topic

[Spectre AMS Designer Analysis Support](#)

Transient Analysis Statement (tran)

The `tran` statement computes the transient response of a circuit over the interval from start to stop. The simulator uses the mixed-signal DC steady-state solution as the initial condition unless you use the `ic` keyword to specify initial conditions. You can specify multiple transient analysis statements in your analog simulation control file.

```
Name tran parameter=value { parameter=value }
```

Parameters

You can use the following parameters and values with the `tran` statement. Values listed in the parameter syntax are the defaults. Unless specifically noted in the definition, you can use these parameters with the Spectre solver only.

Parameter	Definition
annotate=title	Degree of annotation. Values: no, title, sweep, status, and steps
	no Suppresses information about the tran analysis
	title Produces annotation information such as the number of transient analysis steps and how long the analysis takes
	sweep
	status
	steps
cmin=0 F	Minimum capacitance from each node to ground.
compression=all	Turns on data compression. With data compression, the Spectre simulator writes output data for a signal only when the value of that signal changes. This reduces the size of the transient analysis output file for circuits with substantial amounts of signal latency, such as mixed analog and digital designs and circuits with switching power supplies. Values: all, no, wildcardonly If the <code>compression</code> parameter is set to <code>wildcardonly</code>, and the TCL probe statement contains signals specified with a wildcard or a scope, then the signals specified with the wildcard or the scope are compressed. <pre>probe -create -database waves -all -depth all {testbench.I1.*} probe -create -database waves -all -depth all {testbench.I2} probe -create -database waves {testbench.I3.net1} probe -create -database waves {testbench.I4.net2}</pre> In the above example, only the <code>testbench.I1.*</code> and <code>testbench.I2</code> signals will be compressed.
compfactor=1.0	Compression factor. Limits the tolerance values during compressing transient waveforms. The compression decision is made according to <code>tolerance * compfactor</code>.
errpreset=moderate	Selects a reasonable collection of parameter settings. Values: conservative, moderate, liberal
fastbreak=no	Specifies the evaluation method to use for VHDL-AMS <code>break</code> statements. Values:
	no Complies strictly with the VHDL-AMS standard
	yes Requests an often faster method, which under some circumstances does not comply with the VHDL-AMS standard Possible non-compliance with the standard arises when the software associates the break statement with a discontinuity that causes a zero-delay <code>Q'above</code> event. The software might report the <code>Q'above</code> event with a small delay, rather than with a zero delay.

Spectre AMS Designer and Xcelium Simulator Mixed-Signal User Guide

Controls for the Analog Solvers

fastcross=cm	Controls whether the AMS Designer simulator uses an optimized cross-detection algorithm for the @above and @cross functions in Verilog-AMS and the ABOVE function in VHDL-AMS. Values:	
	no	Turns off the optimized algorithm, which you might want to do to verify the behavior and accuracy of models
	discrete	Provides better simulation performance by ignoring the unsatisfiable cross tolerance conditions due to discrete or discontinuous signals in the cross signal expression
	cm	Includes the features of fastcross=discrete and further provides better simulation performance by improving the performance of @above and @cross in connect modules (CMs) such as the ConnRules Possible non-compliance with the standard arises when the software associates the break statement with a discontinuity that causes a zero-delay Q'_{above} event. The software might report the Q'_{above} event with a small delay, rather than with a zero delay. CMs that Cadence provides
	Default Values: (You can override these default values using the fastcross option.) When errpreset=conservative for the Spectre solver, the default is no. When errpreset=liberal or moderate for the Spectre solver.	
flushofftime (s)	Time to stop flushing outputs.	
flushpoints	Flush outputs after number of calculated points.	
flushtime (s)	Flush outputs after real time has elapsed.	
ic=all	What should be used to set initial condition. Values: dc, node, dev, all	
infoname= inf_anal_name	Name of the info analysis to be performed at each time point specified by the infotimes parameter.	
infotimes=[...] s	Times when the simulator is to perform the info analysis specified by the infoname parameter. Setting infotimes to [0] returns the initial transient operating point. Setting infotimes to [stop_time] returns the final transient operating point. If no value is specified for infotimes, the Spectre solver performs the analysis at time 0.	
acnames=[...]	Names of analyses to be performed at each time point in the actimes array. The named small-signal analyses are not run separately, but only as part of the transient analysis.	
actimes=[...]	Times when analyses specified in acname array are performed.	
lteratio	Ratio used to compute LTE tolerances from Newton tolerance. Default derived from errpreset.	
maxiters=5	Maximum number of iterations per time step.	
maxstep (s)	Maximum time step. Default derived from errpreset.	
method	Integration method. Default derived from errpreset. Values: euler, trap, traponly, gear2, gear2only, trapgear2	
oppoint=no	Should operating point information be computed for initial timestep, and if so, where should it be sent. Values: no, screen, logfile, rawfile	
outputstart= time	Output is saved only after this time is reached.	
readic	File that contains the initial conditions.	
readns	File that contains the estimate of initial transient solution.	
relref	Reference used for the relative convergence criteria. Default derived from errpreset. Values: pointlocal, alllocal, sigglobal, allglobal	

save	Signals to output. Values: <code>all</code>, <code>selected</code>, <code>none</code> Cadence recommends using the <code>Tcl probe</code> command to save signals instead of the <code>save</code> command since it has fewer restrictions.
skipcount	Save only one of every skipcount points.
skipdc=no	If <code>yes</code>, there will be no DC analysis for transient. If the DC analysis is skipped, the initial solution is either trivial, or given in the file that you specify using the <code>readic</code> parameter. If the <code>readic</code> parameter is not specified, the values specified on the <code>ic</code> statements are considered. Device-based initial conditions are not used for <code>skipdc</code>. Nodes that you do not specify with the <code>ic</code> file or <code>ic</code> statements start at zero. You should not use this parameter unless you are generating a nodeset file for circuits that have trouble in the DC solution. Values: <code>no</code>, <code>yes</code>, <code>waveless</code>, <code>rampup</code>, <code>autodc</code> Using the AMS Designer simulator, setting <code>skipdc</code> to anything other than <code>no</code> can cause incorrect signal transitions between analog and digital during transient analysis.
skipstart= starttime s	The time to start skipping output data.
start=0 s	Start time.
stats=no	Analysis statistics. Values: <code>no</code> , <code>yes</code>
step=0.001 (stop-start) s	Minimum time step used by the simulator solely to maintain the aesthetics of the computed waveforms.
stop (s)	Stop time.
strobedelay=0 s	The delay (phase shift) between the skipstart time and the first strobe point.
strobeperiod (s)	The output strobe interval (in seconds of transient time).
title	Analysis title.
transres=resolution s	Duration of a transition below which the simulator stops trying to determine exact times for each corner of a transition and starts handling the two corners as a single event.
write	File to which you want the software to write the initial transient solution.
writefinal	File to which you want the software to write the final transient solution.

Multiple transient analyses are not supported in AMS Designer.

Examples

You run a simulation that uses an analog simulation control file with the following contents.

```
myopt options rawfmt=psfbin
myinfo info what=oppoint where=rawfile
mytran tran stop=10u infoname=myinfo infotimes=[1u 2u 3u]
```

The above statements store the operating point data in three different files: `timeDom-000_myinfo.info` (time:1us), `timeDom-001_myinfo.info` (time:2us), `timeDom-002_myinfo.info` (time:3us). All three files are placed in the `.raw` directory.

You run a simulation that uses an analog simulation control file with the following contents.

```
myopt options rawfmt=psfbin
myinfo info what=oppoint where=rawfile
```

The above statements store the DC operating point data in a file called `myinfo.info` located in the `.raw` directory.

Related Topics

- [Mixed-Signal DC Initialization](#)
- [Transient Analysis Statement \(tran\)](#)

Monte Carlo Analysis

The AMS Designer simulator allows you to perform the Monte Carlo analysis on a design to view the impact of a change in analog circuit or model parameters on the design simulation. The AMS Designer Monte Carlo analysis utilizes the native Monte Carlo engine of Spectre to control the AMS (`xmsim`) simulation flow of multiple simulation runs under parameter variations in a single `xmsim` invocation.

The `distribute` parameter of the Monte Carlo analysis not supported in AMS Designer.

In the AMS Designer simulator, you use Tcl commands to control (with stimulus) and monitor (with probes) the simulation. In each iteration of the Monte Carlo analysis, the simulation stimulus in the Tcl input file specified with the `xrun` or `xmsim` command is re-opened and re-applied to the simulation. The probes in the Tcl file are redirected to a new waveform file, which has an iteration-indexed name derived from the originally opened waveform file.

AMS Designer also supports DC analysis inside the Monte Carlo analysis.

Monte Carlo analysis is also supported for VHDL-AMS models and for the AMS-APS flow.

Related Topic

- [Spectre AMS Designer Analysis Support](#)

EMIR Analysis in Spectre AMS Designer

The AMS Designer simulator supports a high performance and high capacity dynamic and static EMIR (Electro-Migration and Voltage drop) analysis which provides a set of advanced features covering power gate support, design resistor EM calculation, macro model generation, black boxing, what-if analysis, and static EMIR analysis.

You can run EMIR analysis as follows:

```
xrun -spectre_args +aps[=liberal|moderate|conservative] +emir=emir.conf netlist.vams ...  
xrun -spectre_args ++aps +emir=emir.conf netlist.vams ...
```

EMIR analysis is enabled by adding the `+emir` option to `-spectre_args`. All EMIR-related commands are defined in the EMIR control file.

Currently, only the APS solver supports EMIR analysis.

Since EMIR analysis is used to perform postlayout simulation, it is applied only to the device-level components in the design. It is not applied to behavioral components.

When you perform EMIR analysis using the AMS Designer simulator with APS solver, the following differences apply:

- The hierarchy in an AMS design is not the same as Spectre
- EMIR analysis is not supported with multiple transient analyses statements in MDL flow
- `altergroup` statements are not supported
- UTI statements are not supported
- IE nets, such as L2E and R2E are treated as signal nets, and not power nets
- Both pre-layout module definition and its instantiation should be in the schematic when using `dspf_include`

Related Topics

- [EMIR Analysis](#)
- [Spectre AMS Designer Analysis Support](#)

Reliability Analysis in Spectre AMS Designer

Reliability analysis computes the reliability for MOSFETs and the circuit. The three reliability modules are:

- MOSFET Hot-Carrier Injection (HCI) module:
Predicts transistor and circuit performance degradation due to HCI effects
- PMOSFET Negative Bias Temperature Instability (NBTI) module:
Predicts PMOSFET and circuit performance degradation due to NBTI and NBTI recovery effects
- NMOSFET Positive Bias Temperature Instability (PBTI) module:
Predicts NMOSFET and circuit performance degradation due to PBTI effects

You can specify the reliability analysis statement in your analog simulation control file as follows:

```
name reliability {
    <reliability control statements> ...
    <stress simulation statements> ...
    <aging testbench statements> ...
    <aging/post-stress simulation statements> ...
}
```

You can specify all analyses supported by AMS in the reliability analysis block. Transient analysis is the only valid analysis type for stress simulation.

The AMS reliability analysis supports SST2 and SST2+PSFXL waveform formats. However there is some limitation to using the analyses types with the waveform format. The following table shows the analysis that you can use in the reliability block based on the waveform format specified:

Analysis Name	SST2	SST2+PSFXL
tran	Yes	Yes
info	Yes	No
ac	Yes	No
alter	Yes	Yes
dc	Yes	No

As shown in the above table, when SST2 format is used, the `tran`, `info`, `ac`, `alter`, and `dc` analyses can be specified in reliability block. However, when SST2+PSFXL format is used, only `tran` and `alter` analyses can be specified in the reliability block.

Related Topics

- [Reliability Analysis](#)
- [Spectre AMS Designer Analysis Support](#)

Checklimit Analysis in Spectre AMS Designer

A `checklimit` analysis allows the enabling or disabling of individual or group of `asserts` specified in the netlist. Use this analysis in conjunction with the `assert` statements in the netlist to perform checks on parameters of device instances, models, subcircuits or expressions. Multiple `checklimit` analyses can be defined in the netlist. The enabled checks will be applied to all subsequent analyses until the next `checklimit` analysis is encountered.

```
Name checklimit parameter=value ...
```

Related Topics

- [Checklimit Analysis \(checklimit\)](#)
- [Spectre AMS Designer Analysis Support](#)

DC Analysis in Spectre AMS Designer

DC analysis finds the DC operating-point or DC transfer curves of the circuit. To generate transfer curves, you need to specify a parameter and a sweep range. The swept parameter can be circuit temperature, a device instance parameter, a device model parameter, a netlist parameter, or a subcircuit parameter for a particular subcircuit instance.

After the analysis is complete, the modified parameter returns to its original value.

The primary difference between Spectre AMS Designer and Spectre DC analysis is that digital simulation and analog/digital synchronization has to be considered in AMS. For DC convergence on analog/digital simulation iteration:

- Analog/Digital simulation at t=0 starts independently from the initial state
- Analog and digital exchange analog/digital signals at each iteration
- DC analysis converges when there are no new analog/digital signals (no new analog-to-digital and digital-to-analog events)

Spectre AMS Designer Simulator supports the following DC analysis parameters:

- State-file Parameters - `readns`, `write`, `writefinal`
- Output Parameters - `save`, `nestlvl`, `print`, `oppoint`
- Convergence Parameters - `homotopy`, `maxiters`, `maxsteps`
- Annotation Parameters - `annotate`

You can run multiple DC analyses at different sweep points. Then, at every sweep point, analog/digital simulation and synchronization iteration is executed. The iteration ends when no new analog-to-digital and digital-to-analog events are detected and the DC analysis converges, or the maximum number of iterations is reached at the sweep point. The table below shows the default value of the maximum iteration number for different solvers.

Solver	setenv AMS_DC_MAX_ITER (default value)
Spectre	100
APS	100
Fast APS (++aps)	100
XPS MS	1

fastdc Option

To speed up DC simulations, you can use the `fastdc` option. The solution accuracy depends on the value specified for `fastdc`. The available values are 0, 1, 2, 3, and 4 (default).

Value	Description
<code>+fastdc=0</code>	The most accurate DC result and closest to true DC solutions with the slowest performance. It should be used for the circuits that require accurate DC solution, such as analog circuits.
<code>+fastdc=1</code>	The conservative DC result with slow performance. It should be used for the circuits that are sensitive to DC solution, such as analog dominated circuits.
<code>+fastdc=2</code>	The reasonable DC result with moderate performance. It should be used for the circuits that are lightly sensitive to DC solution, such as mixed-signal circuits.
<code>+fastdc=3</code>	The liberate DC results with fast performance. It should be used for the circuits that are less sensitive to DC solution, such as digital dominated circuits.
<code>+fastdc=4</code>	The least accurate DC result with the fastest performance. It should be used for the circuits that are not sensitive to DC solution, such as digital and memory circuits.

You can also use `+msdc=ms`. It is the fastest DC algorithm. It is valuable to speed up for large design power up test bench.

dcopt Option

The `dcopt` option speeds up the DC simulation in Spectre and APS for post-layout circuits that consist of large number of parasitic resistors and capacitors, which require significant time in DC simulation. It can also be used to solve the non-convergence issues in DC simulations of any other circuit.

In some cases, the DC solution obtained using the `dcopt` command-line option may not be as accurate as the true DC solution.

Related Topics

- [DC Analysis](#)
- [Spectre AMS Designer Analysis Support](#)

Sweep Analysis (sweep)

The `sweep` analysis sweeps a parameter, running the list of analyses (or multiple analyses) for each value of the parameter. The swept parameter can be circuit temperature, a device instance parameter, a device model parameter, a netlist parameter, or a subcircuit parameter for a particular subcircuit instance. A set of parameters can be swept simultaneously, using the `paramset` parameter. The other sweep interval or variable parameters cannot be specified with the `paramset` parameter.

```
Name sweep parameter=value ...
```

Related Topics

- [Sweep Analysis \(sweep\)](#)
- [Spectre AMS Designer Analysis Support](#)

Stability Analysis (stb)

The `stb` analysis linearizes the circuit about the DC operating point and computes the loop gain and gain and phase margins (if the sweep variable is frequency) for a feedback loop or a gain device. Spectre can perform the analysis while sweeping a parameter. The parameter can be frequency, temperature, component instance parameter, component model parameter, or netlist parameter. If changing a parameter affects the DC operating point, the operating point is recomputed at each step.

```
Name stb parameter=value ...
```

For more information on `stb` analysis parameters, refer to the *Stability Analysis (stb)* section in the *Spectre Circuit Simulator Reference* manual.

Related Topics

- [Stability Analysis \(stb\)](#)
- [Spectre AMS Designer Analysis Support](#)

Transfer Function Analysis (xf)

The transfer function analysis linearizes the circuit about the DC operating point and performs a small-signal analysis that calculates the transfer function from every independent source in the circuit to a designated output. The variable of interest at the output can be voltage or current.

You can specify the output with a pair of nodes or a probe component. Any component with two or more terminals can be a voltage probe. When there are more than two terminals, they are grouped in pairs, and you use the `portv` parameter to select the appropriate pair of terminals. Alternatively, you can specify a voltage to be the output by giving a pair of nodes on the `xf` analysis statement.

Any component that naturally computes current as an internal variable can be a current probe. If the probe component computes more than one current (as transmission lines, microstrip lines, and N-ports do), you use the `porti` parameter to select the appropriate current. It is an error to specify both `portv` and `porti`. If neither is specified, the probe component provides a reasonable default.

The `stimuli` parameter specifies the inputs for the transfer functions. There are two choices. `stimuli=sources` indicates that the sources present in the circuit should be used. The `xfmag` parameters provided by the sources may be used to adjust the computed gain to compensate for gains or losses in a test fixture. You can limit the number of sources in hierarchical netlists by using the `save` and `nestlvl` parameters.

```
Name [p] [n] xf parameter=value ...
```

The optional terminals (`p` and `n`) specify the output of the circuit. If you do not specify the terminals, you must specify the output with a probe component.

Related Topics

- [Transfer Function Analysis \(xf\)](#)
- [Spectre AMS Designer Analysis Support](#)

Initial Conditions (ic) Statement

The `ic` statement specifies initial conditions for nodes in the transient analysis. If you have more than one `ic` statement, the simulator collects information for all occurrences. You can specify initial conditions for the following items:

- Inductor currents
- Node voltages where the nodes have a path of capacitors to ground

```
ic node=value { node=value }
```

Parameter	Definition
<code>node=value</code>	Important: For AMS simulation, you must always include the top cell name in the <code>node</code> specification. Each <code>node</code> is a topological node of the circuit or an unknown, such as the current through an inductor or the voltage of the internal node in a diode. Topological nodes can be at the top level or in a subcircuit.

For example, the statement

```
ic top.n7=0 top.OpAmp1.comp=5 top.L1:t1=1.0u v(top.n1)=1.5
```

specifies that

- Node `n7` has an initial value of 0 V,
- Node `comp` in subcircuit `OpAmp1` has an initial value of 5 V,
- The current through the first terminal of `L1` has an initial value of 1 uA, and
- Node `n1` has an initial value of 0.5 V.

Both the solvers (Spectre and APS) support `node=value` syntax as well as `v(node)=value` syntax.

Related Topics

- [Initial Conditions \(ic\)](#)
- [Spectre AMS Designer Analysis Support](#)

alter Statement

The `alter` statement changes the value of any modifiable component or netlist parameter for any analyses that follow. The parameter to be altered can be circuit temperature, a device instance parameter, a device model parameter, a netlist parameter, or a subcircuit parameter for a particular subcircuit instance.

```
Name alter parameter=value ...
```

Alter Group (altergroup)

The `altergroup` statement changes the values of any modifiable model, instance or netlist parameter for any analyses that follow. Within an alter group, you can specify model statements, instance statements, parameter statements and options statements (only supports temp, tnom, and scale). These statements should be bound within braces. The opening brace is required at the end of the line defining the altergroup. Altergroups cannot be specified within subcircuits. The `analyses`, `export`, `paramset`, `save`, and `sens` statements are not allowed within altergroups.

```
Name altergroup parameter=value ...
```

Related Topics

- [Alter a Circuit, Component, or Netlist Parameter \(alter\)](#)
- [Alter Group \(altergroup\)](#)
- [Spectre AMS Designer Analysis Support](#)

Device Checking and Violations Display

In the AMS Designer simulator, the device checking feature works exactly the way it works in Spectre. You can use `assert` statements in your netlist to set custom characterization checks that specify the safe operating conditions for your circuit. The `assert` statements are supported for transient, AC, and DC analyses.

The violation data generated by the `assert` statements and the associated `checklimit` analysis is reported at the command line and is also displayed in an analog design environment (Virtuoso ADE) GUI.

Related Topic

- [assert Statement](#)

info Statement

The `info` statement outputs several kinds of information about circuits and components. You can use various filters to specify what information is output. You can create a listing of model, instance, temperature-dependent, input, output, and operating-point parameters. You can generate a summary of the minimum and maximum parameter values, and you can request that the simulator provide a node-to-terminal map or a terminal-to-node map.

When the `info` statement precedes the transient analysis statement, the simulator performs the `info` analysis at time zero before the transient analysis begins. When the `info` statement follows the transient analysis statement, the simulator performs the `info` analysis after the transient analysis finishes.

```
name info parameter=value { parameter=value }
```

For `parameter`, you can substitute `what`, `where`, `file`, `extremes`, `save`, `title`, and `writedc` as described in the following sections.

what

The `what` parameter of the `info` statement specifies what parameters are to be output. The values that you can use with the `what` parameter are listed in the following table. Unless specifically noted in the Action column, these parameters are supported only by the Spectre and APS solvers.

Settings	Action
<code>all</code>	Outputs a list of input and output parameter values.
<code>input</code>	Outputs information about inputs.
<code>inst</code>	Outputs information about instances. .
<code>models</code>	Outputs information about models. (This setting is effective only for Verilog [®] -AMS.)
<code>nodes</code>	Outputs a terminal-to-node map.
<code>none</code>	Does not print any parameters.
<code>oppoint</code>	Outputs device parameter information about DC operating points. This is the default. To request node voltage output during an operating point analysis, specify the <code>info writedc</code> parameter.
<code>output</code>	Outputs information about outputs. (This setting is effective only for Verilog-AMS.)
<code>parameters</code>	Outputs netlist parameter values such as <code>temp</code> and <code>tnom</code> . (This setting is effective only for Verilog-AMS.)
<code>subckts</code>	Outputs subcircuit parameters.
<code>terminals</code>	Outputs a node-to-terminal map.

where

The `where` parameter of the `info` statement specifies a destination for the information you want the simulator to output. You can specify the following `where` parameter settings whether you are using the AMS Designer simulator with the Spectre solver or the APS solver:

Settings	Action
<code>file</code>	Sends the output to a file that you create. If you use this setting, use the <code>file</code> parameter to specify the output file.
<code>logfile</code>	Sends the parameters to the simulator log file. This is the default.
<code>nowhere</code>	Produces no output.
<code>rawfile</code>	Sends the output to rawfile in the format specified by the <code>options</code> statement <code>rawfmt</code> parameter. The rawfile is <code>.raw/myinfo.info</code> where <code>myinfo</code> is the name of the <code>info</code> statement.
<code>screen</code>	Displays the output on a screen.

file

The `file` parameter of the `info` statement specifies where the values of the `info` analysis are saved when the `where` parameter of the `info` statement is set to `file`. The `file` parameter is supported only by the Spectre solver and the APS solver.

Settings	Action
<code>"filename"</code>	Specifies a file to receive the output requested by the <code>info</code> statement. This is the file that is used when you specify the <code>where=file</code> parameter and setting. The default value is <code>scs.info.what_setting</code> where <code>scs</code> is the basename of the analog simulation control file that specifies the command, and <code>what_setting</code> is the active value for the <code>info -what</code> parameter.

extremes

The `extremes` parameter of the `info` statement generates a summary of maximum and minimum parameter values. The `extremes` parameter is supported only by the Spectre solver and the APS solver.

The values that you can use with the `extremes` parameter are listed in the following table.

Settings	Action
<code>no</code>	Does not generate a summary of minimum and maximum parameter values.
<code>only</code>	Generates a summary of only minimum and maximum values.
<code>yes</code>	Generates a summary of minimum and maximum values, including information on the other values that contribute to the extreme values. This is the default.

save

The `save` parameter of the `info` statement specifies signals to save during a transient or AC analysis, or during a DC analysis specified by an `info writedc` statement. The `save` parameter is supported only by the Spectre solver and the APS solver.

You values that you can use with the `save` parameter are listed in the following table.

Settings	Action
<code>all</code>	Saves all signals.
<code>selected</code>	Saves only signals specified with <code>save</code> statements.
<code>none</code>	Saves all signals.

title

The `title` parameter of the `info` statement specifies a name for the analysis. The `title` parameter is supported only by the Spectre solver and the APS solver.

Settings	Action
"title"	Specifies a subheading for the analysis. The subheading appears at the top of the analysis output, just below the name of the analysis.

writedc

The `writedc` parameter of the `info` statement outputs the DC node voltages in [PSF](#) format. The `writedc` parameter is supported only by the Spectre solver and the APS solver.

Settings	Action
"filename"	Specifies the basename for a file into which the DC node voltage values are written in PSF format. The generated file has the name <code>filename.dc</code> .

Examples

You run a simulation that uses an analog simulation control file called `sch.scs` with the following contents.

```
simulator lang=spectre
PrintInputParams info what=input where=file
tran1 tran stop=14us errpreset=moderate maxiters=10 cmin=10f
    infoname=PrintOppoint infotimes=[50n 150n]
PrintNodesParams info what=nodes where=file file="./nodes.txt"
    title= "Nodes and Values"
PrintOppoint info what=oppoint where=file file="./oppoint.txt"
```

This example illustrates some of the combinations that you can specify.

- The `PrintInputParams` analysis is specified before the transient analysis so the `what=input` values are determined prior to the start of the transient analysis. The information is to be written to a default file, which, in this case, has the name `sch.info.input`, because the analog simulation control file is called `sch.scs` and the `what` parameter has the value `input`.
- The `infotimes` parameter on the `tran1` statement specifies that the analysis specified by the `infoname` parameter (`PrintOppoint`) is to run twice during the transient analysis, at 50 nanoseconds and 150 nanoseconds. The `PrintOppoint` analysis is specified by the last line in the analog simulation control file.
- The `PrintNodesParams` information is to be written to the file specified by the `file="./nodes.txt"` value. The listing is to have the subheading `Nodes and Values` so that the result looks like this:

```
*****
Terminal to Node Map `PrintNodesParams'
*****
Nodes and Values
```

The `PrintNodesParams` information is determined after the transient analysis ends because the `PrintNodesParams` analysis is specified after the transient analysis is specified.

Related Topics

- [Mixed-Signal DC Initialization](#)

save Statement

The `save` statement specifies nodes or signals whose values are to be saved in the output file. This statement supports saving only the voltage of only analog signals.

```
save signal { signal }
```

The `signal` that you specify must be the complete hierarchical name.

Be careful when you specify a signal name that contains language-specific special characters. The name you use on the save statement must be legal in the Spectre language and must map to a name that is legal in the Verilog-AMS language.

The format for the saved signal waveform is determined by the `options rawfmt` parameter. The location of the database for the saved signal waveform is determined by the `options rawfile` parameter.

For example, the following statement specifies that the AMS Designer simulator is to save the signal `sig` found inside the hierarchy specified by `top.il`.

```
save top.il.sig
```

The next example illustrates a name mapping complication.

```
save cds_globals.\\GND!\\ // There is a space after the last backslash.
```

The ``cds_globals.\\GND!`` name, when mapped to Verilog-AMS, produces ``cds_globals.\\GND!``, which is in the required format.

In AMS Flex flow, the ground node does not appear in waveform database as the ground node is always 0 Volt. This is consistent with the Spectre simulator behavior.

Related Topics

- [The save Statement](#)
- [Spectre AMS Designer Analysis Support](#)

print Statement

For the AMS Designer simulator using the Spectre solver with the Simulation Front End (SFE) parser, you can use the Spectre language `print` statement to print signal and instance data to an output file for AC and transient analyses. For detailed information about the `print` statement, see "Print Statement" in the "Spectre Netlists" chapter of the Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide.

The output directories created by commands `-xmlibdirname`, `-xmlibdirpath`, `+amsrawdir`, and `-outdir` specified using the solver arguments are redirected to user-specified directories instead of the default directories.

Related Topics

- [The print Statement](#)
- [Spectre AMS Designer Analysis Support](#)

Mixed-Signal DC Initialization

The Spectre AMS Designer Simulator features a consistent mixed-signal DC initialization for the following analyses using all supported languages in all simulation flows:

- Info
- AC
- Transient

Supported languages include SPICE, Spectre, Verilog-A, Verilog-AMS, and VHDL-AMS. Supported flows include the analog design environment (ADE), the AMS Designer environment, and AMS in verification.

Mixed-signal DC is an AMS simulation process that iterates between analog DC analysis and digital simulation at time 0 until all signals at the analog or digital boundary reach steady state. Any info, AC, or transient analysis you request in your analog simulation control file begins with a mixed-signal DC initialization, whether you request it explicitly (using `oppoint`, for example) or implicitly (by specifying a transient analysis, for example).

With mixed-signal DC initialization, you get consistent simulation results when you change a module's view (from Verilog to SPICE, for example). The AMS designer simulator correctly reflects the digital signal (stimulus) of a Verilog view in the behavior of analog analyses such as DC, AC, and transient. For example, you can have a design that has a digital Verilog module feeding a Verilog-A or SPICE subcircuit of an analog filter where the digital signal

controls the pole and zero positions of the analog filter.

The AMS Designer simulator with Spectre solver executes the code inside `@initial_step` during all analog or digital DC iterations for mixed-signal initialization of transient analysis.

The default maximum number of mixed-signal DC iterations differs depending on which solver you are using as follows:

Solver	Default Maximum Number of Mixed-Signal DC Iterations
Spectre	100
APS	100

You can control the maximum number of mixed-signal DC iterations by setting the `AMS_DC_MAX_ITER` environment variable as follows:

```
setenv AMS_DC_MAX_ITER = numIterations
```

For example, to specify a maximum of five iterations,

```
setenv AMS_DC_MAX_ITER = 5
```

True DC analysis is currently not available.

Related Topics

- [AC Analysis Statement \(ac\)](#)
- [Transient Analysis Statement \(tran\)](#)

Comparing the Behavior of save Parameter Specified in an Analysis and the Global save Option

If you have the `save` parameter specified in an analysis statement (for example, `tran`, `ac`, and so on), the `save` parameter setting specified for that analysis overrides the `save` option specified using the global `options` statement. However, if `save = none` is specified in an analysis statement, then, the value specified using the global `options` statement is considered. The following table shows the behavior of the `save` parameter when specified in both transient analysis and global `options` statement.

tran Statement	options Statement	Behavior
<code>save=none</code>	<code>save=none</code>	Saves any random voltage signals.
<code>save=selected</code>	<code>save=none</code>	Saves only selected signals specified using the <code>save</code> statement in the netlist.
<code>save=none</code>	<code>save=selected</code>	Uses <code>save=selected</code> specified in the global options statement.
<code>save=selected</code>	<code>save=selected</code>	Saves only selected signals specified using the <code>save</code> statements in the netlist.

Related Topic

- [The save Statement](#)

Circuit Checks for Spectre AMS Designer

With AMS Designer, you can perform static and dynamic checks. The following table specifies the static checks supported in AMS Designer.

Static Circuit Check	Description
Static Capacitor Check (<code>static_capacitor</code>)	Reports all capacitors within or outside the range of <code>cmin</code> and <code>cmax</code> .

Spectre AMS Designer and Xcelium Simulator Mixed-Signal User Guide
Controls for the Analog Solvers

Static Capacitor Voltage Check (<code>static_capv</code>)	Reports capacitor devices fulfilling the conditional expression on device voltages.
Static Coupling Impact Check (<code>static_coupling</code>)	Evaluates the possible coupling effects in the circuit.
Static DC Leakage Path Check (<code>static_dcpath</code>)	Reports the always conducting paths between the power supply nodes.
Static Diode Voltage Check (<code>static_diodev</code>)	Reports the diode devices fulfilling the conditional expression on device voltages.
Static ERC Check (<code>static_erc</code>)	Performs various electrical rule checks and reports the devices with violations.
Static HighZ Node Check (<code>static_highz</code>)	Reports the nodes that do not have any possible conducting path to a DC power supply or ground.
Static MOSFET Voltage Check (<code>static_mosv</code>)	Reports the MOSFET devices fulfilling the conditional expression on device voltages and device size (w, l).
Static NMOS to vdd count (<code>static_nmos2vdd</code>)	This check counts the unpaired NMOS in a charging path from a fanout node to VDD and report paths that have an NMOS count greater than specified count.
Static NMOS Forward Bias Bulk Check (<code>static_nmosb</code>)	Reports the NMOS devices with a forward-biased bulk condition.
Static Always Conducting NMOSFET Check (<code>static_nmosvgs</code>)	Reports the NMOS devices that are potentially always conducting due to connectivity problems.
Static PMOS to gnd count (<code>static_pmos2gnd</code>)	This checks counts the unpaired PMOS in a discharging path from the fanout node to GND and report paths that have a PMOS count greater than the specified count.
Static PMOS Forward Bias Bulk Check (<code>static_pmosb</code>)	Reports the PMOS devices with a forward-biased bulk condition.
Static Always Conducting PMOSFET Check (<code>static_pmosvgs</code>)	Reports the PMOS devices potentially always conducting due to connectivity problems.
Static RCDelay Check (<code>static_rcdelay</code>)	Reports nodes with excessive rise or fall times. Rise and fall times are based on estimation.
Static Resistor Check (<code>static_resistor</code>)	Reports all resistors within or outside the range of <code>rmin</code> and <code>rmax</code> . It can also generate a distribution list of all the resistors in a circuit.
Static Resistor Voltage Check (<code>static_resv</code>)	Reports the resistor devices fulfilling the conditional expression on device voltages.
Static Subckt Port Voltage Check (<code>static_subcktpport</code>)	Reports instances of the user-specified subckt fulfilling the conditional expression on port voltages.
Static Transmission Gate Check (<code>static_tgate</code>)	Reports the transmission gates which cause potential leakage currents between power supplies.
Static Voltage Domain Conflict Check (<code>static_vconflict</code>)	Reports low voltage domain driving MOSFET devices from high voltage domain.
Static Voltage Domain Device Check (<code>static_voltdomain</code>)	Reports the high voltage driving the low-voltage MOSFET and low voltage driving the high-voltage MOSFET.

The following table specifies the dynamic checks supported in AMS Designer.

Dynamic Circuit Check	Description
Dynamic Subckt Instance Activity Check (<code>dyn_activity</code>)	Reports activity percentage of an instance relative to a circuit.

Dynamic Active Node Check (<i>dyn_actnode</i>)	Detects nodes with voltage changes that exceed the user-defined threshold <i>dv</i> .
Dynamic Capacitor Voltage Check (<i>dyn_capv</i>)	Reports capacitor elements fulfilling the conditional expression on element voltages for a duration longer than the user-specified threshold duration.
Dynamic DC Leakage Path Check (<i>dyn_dcpath</i>)	Reports conductance paths between user-specified nets.
Dynamic Delay Check (<i>dyn_delay</i>)	Checks timing delays between two signals and reports nodes with edge delay errors.
Dynamic Diode Voltage Check (<i>dyn_diodev</i>)	Reports diode elements fulfilling the conditional expression on element voltages for a time longer than a user-specified duration threshold.
Dynamic Excessive Element Current Check (<i>dyn_exi</i>)	Reports elements and devices carrying currents (absolute value) higher than the current threshold (<i>ith</i>) for a time longer than the duration threshold (<i>duration</i>).
Dynamic Excessive Rise, Fall, Undefined State Time Check (<i>dyn_exrf</i>)	Reports the nodes with excessive rise times, fall times, or with an undefined state.
Dynamic Floating Node Induced DC Leakage Path Check (<i>dyn_floatdcpath</i>)	Reports the DC leakage paths that are caused by floating nodes.
Dynamic Glitch Check (<i>dyn_glitch</i>)	Checks the signal glitches in the blocks with single and constant power supply.
Dynamic HighZ Node Check (<i>dyn_highz</i>)	Reports the nodes that are in high impedance state for a duration longer than the user-defined threshold.
Dynamic MOSFET Voltage Check (<i>dyn_mosv</i>)	Reports MOSFET devices fulfilling the conditional expression on device voltages and device size (<i>w</i> , <i>l</i>) for a time longer than the user-specified threshold duration.
Dynamic Node Capacitance Check (<i>dyn_nodecap</i>)	Reports the node capacitance at specified times (<i>time</i>) of a transient simulation.
Dynamic Noisy Node Check (<i>dyn_noisynode</i>)	Identifies the nodes with unstable or noisy node conditions.
Dynamic Pulse Width Check (<i>dyn_pulsewidth</i>)	Reports nodes with pulse width error.
Dynamic Resistor Voltage Check (<i>dyn_resv</i>)	Reports the resistor elements fulfilling the conditional expression on element voltages for a duration longer than the user-specified duration threshold.
Dynamic Setup and Hold Check (<i>dyn_setuphold</i>)	Reports nodes with setup or hold timing errors with reference to a clock signal. The transition time of the clock signal is <i>refTime</i> .
Dynamic Statistical HighZ Node Check (<i>dyn_stahighz</i>)	Identifies and reports nodes that are in high impedance state at a user given time.
Dynamic Subckt Port Voltage/Current Check (<i>dyn_subcktpport</i>)	Reports instances of the user-specified subckt fulfilling the conditional expression on port voltages and port currents for a time longer than the user-specified <i>duration</i> .
Dynamic Subckt Port Power Check (<i>dyn_subcktpwr</i>)	Reports port currents, port powers, and subcircuit powers.

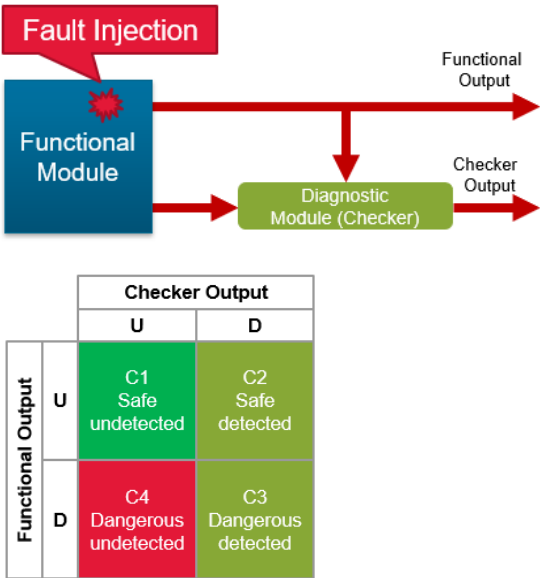
The dynamic checks are not supported in the AMS +ms mode.

Related Topic

- [Circuit Checks](#)

Fault Analysis in Spectre AMS Designer

In the AMS flex mode, you can perform the analog fault analysis of the Legato Reliability Solution. The fault analysis provides a transistor-level simulation capability that can be enabled in an analog test methodology to improve test coverage by identifying critical test patterns. The following figure depicts how fault simulation functions.



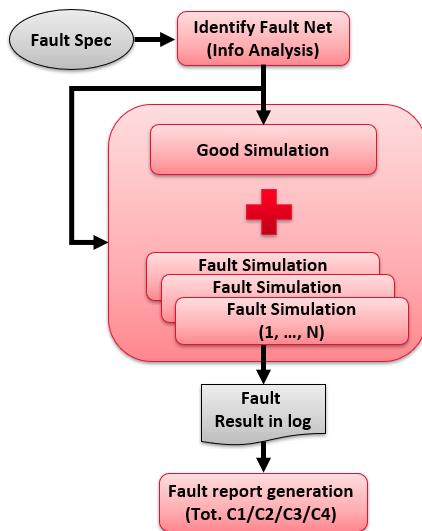
Example:

```
name faults <parameter=value> {
  bridge {
    <name> ( <node1> <node2> ) r = <expression>
  }
  short {
    <name> ( <node1> <node2> ) r = <expression>
  }
  open {
    <name> ( <node> ) r = <expression> { instance1[terminal 1 terminal2 ...] ... }
  }
  custom {
    ...
  }
}
```

For information about the Legato Reliability Solution, refer to the *Legato Reliability Solution* user guide.

Fault Simulation Use Model

The following figure depicts the use-model of the fault simulation.



Fault analysis is done in the following steps:

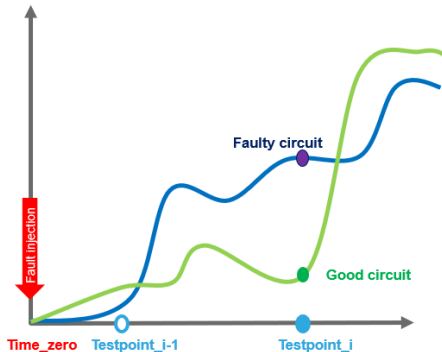
Identifying the fault injection nets - fault injections are applied to device or nets.

Performing good and fault simulation - sweep one good and multiple fault simulations.

Analyzing the simulation results - generation of fault reports.

You can perform the fault analysis in AMS flex mode using one of the following approaches:

Direct Fault analysis: In the direct fault analysis flow, faults are injected at the beginning of each analysis type, such as dc, ac, transient, and so on. This mode provides the most accurate results. However, when the number of faults is large, the simulation takes much longer to complete.



Transient Fault Analysis: To address the challenge of long simulation time, a new algorithm has been implemented in transient fault analysis. In this simulation flow, a circuit with fault list is simulated together with the fault times specified on the time axis. Spectre starts regular transient analysis till the time reaches the fault time point. Then, at each fault time, following actions are performed:

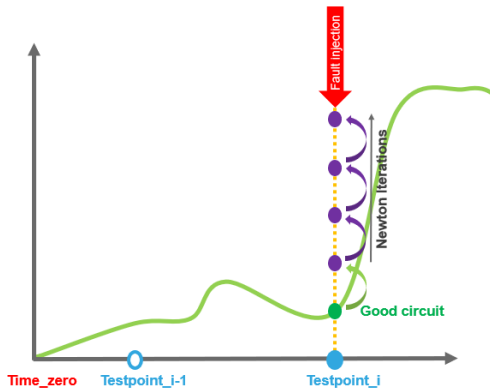
- Fault-free transient state is saved

A fault is injected into the circuit

With fault-free state used as the initial condition, transient fault simulation is continued with regular time step control and accuracy settings

The result is saved and the database is cleaned when all fault solutions are found

Fault-free transient simulation is restored and continues till the next test time point



AMS Designer Spectre Fault Analysis Integration

AMS Designer supports checkers from the digital domain using the `$cds_violation` system function. You can write your own checker modules in Verilog-AMS or SystemVerilog with this system function embedded.

```
$cds_violation ("assertName", "assertMessage", assertType);
```

Here, `assertType` can be 0: None; 1: func, or 2: check). Type 1 and 2 are both for fault-related asserts that will be covered for the functional safety report generation.

AMS Designer supports Spectre functional safety report generation executable file `spectre_fs rpt`. Following is the use model:

```
%../bin/spectre_fs rpt xrun.log -o report_file.txt
```

Note that the program must be executed under the simulation directory.

Currently, AMS Designer does not support fault asserts from both digital side and analog side (device checkers and Verilog-A asserts). When they co-exist, asserts from digital side will take higher priority and analog asserts will be ignored during the functional safety report generation.

Related Topics

- [Run Options for Direct Fault Analysis](#)
- [Run Options for Transient Fault Analysis](#)

nodeset Statement

The `nodeset` statement supplies estimates of solutions that aid convergence or bias the simulation toward a given solution. You can use nodesets for all DC and initial transient analysis solutions. If you have more than one `nodeset` statement in the input, the simulator collects the information.

For AMS simulation, you must always include the top cell name in the `node` specification.

```
nodeset node = value { node = value }
```

Each `node` is a signal. Each signal is a value associated with a topological node of the circuit or with some other unknown that is solved by the simulator. For example, the unknown value might be the current through an inductor or the voltage of the internal node in a diode.

For example, the statement

```
nodeset top.n1=0 top.out=1 top.OpAmp1.comp=5 top.L1:tl=1.0u
```

specifies that

- Node `n1` should be about 0 V,
- Node `o ut` should be about 1 V,
- Node `comp` in subcircuit `OpAmp1` should be about 5 V, and
- The current through the first terminal of `L1` should be about 1u A.

Related Topics

- [The ic and nodeset Statements](#)
- [Spectre AMS Designer Analysis Support](#)

nodeset Statement in Analog Simulation Control File

You use the `nodeset` statement in the analog simulation control file to supply estimates of solutions that aid convergence or bias the simulation towards a given solution. You can use nodesets to provide an initial condition calculation for the transient analysis. The `nodeset` statement has the following format:

```
nodeset signalName=value ...
```

Values you can supply with the `nodeset` statement include voltages on topological nodes, including internal nodes, and currents through voltage sources, inductors, switches, transformers, N-ports, and transmission lines.

For example,

```
nodeset Voff=0 X3.n7=2.5 M1:int_d=3.5 L1:l=1u
```

sets the following solution estimates:

- The voltage of node `Voff` is set to 0.
- Node `n7` of subcircuit `X3` is set to 2.5 V.
- The internal drain node of component `M1` is set to 3.5 V.
- The current for inductor `L1` is set to 1 μ .

Related Topic

- [The ic and nodeset Statements](#)

Spectre or APS Block-Based Transient Noise Analysis

Transient noise provides the benefit of examining the effects of large signal noise on many types of systems. It gives you the opportunity to examine the impact of noise in the time domain on various circuit types without requiring access to the SpectreRF analysis. This capability is an extension to the current transient analysis, and is accompanied by enhancements to several calculator functions, allowing you to calculate multiple occurrences of measurements such as risetime and overshoot.

AMS-Spectre supports only the single run method of simulating transient noise. The single run method involves a single transient run over several cycles of operation. The native Spectre multiple run method is not supported in AMS-Spectre at this time.

The use model adopted by APS for transient noise analysis is identical to the one used by the Spectre circuit simulator.

Set the following parameters to calculate noise during a transient analysis.

Parameter	Definition
<code>noisefmax=0</code> (Hz)	Bandwidth of pseudorandom noise sources. A valid (nonzero) value turns on the noise sources during transient analysis. The maximal time step of the transient analysis is limited to $1/\text{noisefmax}$.
<code>noiseon=[...]</code>	The list of instances to be considered as noisy during transient noise analysis.
<code>noiseoff=[...]</code>	The list of instances to be considered as not noisy during transient noise analysis.
<code>noisescale=1</code>	Noise scale factor applied to all generated noise. It can be used to artificially inflate the small noise to make it visible over transient analysis numerical noise floor, but it should be small enough to maintain the nonlinear operation of the circuit.
<code>noiseseed</code>	Seed for the random number generator. Specifying the same seed allows you to reproduce a previous experiment.
<code>noisefmin</code> (Hz)	If specified, the power spectral density of noise sources depend on frequency in the interval from <code>noisefmin</code> to <code>noisefmax</code> . Below <code>noisefmin</code> , noise power density is constant. The default value is <code>noisefmax</code> , so that only white noise is included and noise sources are evaluated at <code>noisefmax</code> for all models. $1/\text{noisefmin}$ cannot exceed the requested time duration of transient analysis.
<code>noisetmin</code> (s)	Minimum time interval between noise source updates. Default is $1/\text{noisefmax}$. Smaller values will produce smoother noise signals at the expense of reducing time integration step.

<code>noiseupdate=fmax step</code>	Specifies whether noise is to be injected at a constant time step (<code>fmax</code>) or the Spectre solver time step is to be used (<code>step</code>). Injecting noise at a constant time step is suitable when the value of <code>noisefmax</code> is larger than the bandwidth of all signals in the circuit, and simulation time step is effectively controlled by noise. Only one noise frequency is updated at each time step. If the bandwidth of some of the signals exceeds <code>noisefmax</code>, which forces the simulator to take steps smaller than <code>noisetmin</code>, then noise should also be injected at each time step between the regular noise updates. In this case, all noise frequencies are updated at each time step.

Example

```
tr1 tran stop=4u noisefmax=5G noisefmin=1Meg noiseseed=1 noisescale=10 \param=isnoisy param_vec=[0 1 10ns 0 50ns 1]
tr1 tran stop=4u noisefmax=5G noiseupdate=step noiseseed=1 noisescale=10
```

Related Topic

- [Transient Noise Analysis](#)

State Information for Individual Analyses

You can specify state information for individual analyses in two ways:

- You can use the `ic` parameter of the transient analysis to choose which previous specifications are used. You can choose from the following settings:

Parameter setting	Action taken
node	The <code>ic</code> statements are used, and the <code>ic</code> parameter settings on the capacitors and inductors are ignored.
dev	The <code>ic</code> parameter settings on the capacitors and inductors are used, and the <code>ic</code> statements are ignored.
all	Both the <code>ic</code> statements and the <code>ic</code> parameters are used. If specifications conflict, <code>ic</code> parameters override <code>ic</code> statements.

- You can specify initial conditions and estimate solutions by creating a state file that is read by the transient analysis. For example, you can save the solution at the final point of a transient analysis and then continue the analysis in a later simulation by using the state file as the starting point for another transient analysis. You can also use state files to create automatic updates of initial conditions and nodesets. You can instruct the simulator to write a state file from the initial point in an analysis, by using the `write` parameter. The following example writes a state file named `ua741.tran`.

```
timeDom tran stop=1u readns="ua741.tran" write="ua741.tran"
```

You can also instruct the AMS Designer simulator to create a state file in the transient analysis for future use.

Related Topic

- [Initial Conditions \(ic\)](#)

Time-Saving Techniques for the Analog Solvers

This topic discusses different methods to reduce the time devoted to simulating the analog sections of a design.

- [Adjusting Speed and Accuracy](#)
- [Saving Time by Selecting a Continuation Method](#)
- [Specifying Efficient Starting Points](#)

Adjusting Speed and Accuracy

When you use the Spectre solver, you can use the `errpreset` parameter to increase the speed of transient analyses, but this speed increase requires some sacrifice of accuracy. The greatest speedup comes from using `errpreset=liberal`. Greater accuracy, but lower speed, can be obtained by using `errpreset=moderate`, or `errpreset=conservative`.

Saving Time by Selecting a Continuation Method

The Spectre® AMS Designer simulator analog solver normally starts with an initial estimate and then tries to find the solution for an analog circuit using the Newton-Raphson method. If this attempt fails, the simulator automatically tries several continuation methods to find a solution and tells you which method was successful. Continuation methods modify the circuit so that the solution is easy to compute and then gradually change the circuit back to its original form. Continuation methods are robust, but they are slower than the Newton-Raphson method.

If you need to modify and resimulate a circuit that was solved with a continuation method, you probably want to save simulation time by directly selecting the continuation method you know was previously successful.

You select the continuation method with the `homotopy` parameter of the `set` or `options` statements. In addition to the default setting, `all`, you can set the parameter to: `gmin` stepping (`gmin`), source stepping (`source`), the pseudotransient method (`ptran`), and the damped pseudotransient method (`dptran`). You can also prevent the use of continuation methods (`none`).

Specifying Efficient Starting Points

The Spectre AMS Designer Simulator's analog solver arrives at a solution for a simulation by calculating successively more accurate estimates of the final result. You can increase simulation speed by providing state information (the current or last-known status or condition of a process, transaction, or setting) to the transient analysis. You can specify two kinds of state information:

- **Initial conditions**

The `ic` statement lets you specify voltages on nodes for the starting point of a transient analysis. In the Verilog®-AMS source, you can specify voltages on capacitors and currents on inductors.

- **Nodesets**

Nodesets are estimates of the solution you provide for the transient analysis. Unlike initial conditions, nodeset values have no effect on the final results. Nodesets usually act only as aids in speeding convergence, but if a circuit has more than one solution, as with a latch, nodesets can bias the solution to the one closest to the nodeset values.

Setting Initial Conditions

You can specify initial conditions that apply to the transient analysis. The `ic` statement and the `ic` parameter described in this section set initial conditions for the transient analysis in the netlist. In general, you use the `ic` parameter of individual components to specify initial conditions for those components, and you use the `ic` statement to specify initial conditions for nodes. You can specify initial conditions for inductors with either method.

Do not confuse the `ic` parameter for individual components with the `ic` parameter of the transient analysis. The latter lets you select from among different initial condition specifications for a given transient analysis.

- **Specifying initial conditions for components**

You can specify initial conditions in the instance statements of capacitors, inductors, and windings for magnetic cores. The `ic` parameter specifies initial voltage values for capacitors and current values for inductors and windings. In the following Verilog-AMS example, the initial condition voltage on the capacitor is set to two volts:

```
capacitor #(.c(1u),.ic(2)) c1 (net1,net2) ;
```

- **Specifying initial conditions for nodes**

You use the `ic` statement in the analog simulation control file to specify initial conditions for nodes or initial currents for inductors. The nodes can be inside a subcircuit or internal nodes to a component.

The following is the format for the `ic` statement:

```
ic signalName=value ...
```

For example,

```
ic Voff=0 X3.n7=2.5 M1:int_d=3.5 L1:l=1u
```

sets the following initial conditions:

- The voltage of node `Voff` is set to 0.
- Node `n7` of subcircuit `X3` is set to 2.5 V.
- The internal drain node of component `M1` is set to 3.5 V. (See the following table for more information about specifying internal nodes.)
- The current for inductor `L1` is set to 1μ.

Specifying initial node voltages requires some additional discussion. The following table tells you the internal node voltage specifications you can use with

different components.

Component	Internal node specifications
BJT	int_c, int_b, int_e
BSIM	int_d, int_s
MOSFET	int_d, int_s
GaAs MESFET	int_d, int_s, int_g
JFET	int_d, int_s, int_g, int_b
Winding for Magnetic Core	int_Rw
Magnetic Core with Hysteresis	flux

Related Topics

- [Initial Conditions \(ic\) Statement](#)
- [nodeset Statement](#)

amsd Block

An `amsd` block contains statements that control AMS mechanisms during the elaboration phase of `xrun` (or during `xmelab`) in your design verification flow. The simulation front end (SFE) parser reads and applies elaboration statements during the elaboration phase and simulation statements during the simulation phase. `amsd` blocks are valid only for AMS simulation.

You can use statements in an `amsd` block to specify the design configuration, connect module settings, port mappings between SPICE and Verilog, as well as to configure a particular cell reference in a Verilog-AMS source file to be bound to a SPICE description of that cell. These AMS statements must appear only in an `amsd` block.

You can have more than one `amsd` block in one or more Spectre or SPICE source files. You can pass one or more such input files directly on the `xrun` command line. If you pass more than one input file, the SFE parser processes them in the order they appear on the command line.

You can put other Spectre-language commands in the input file as long as you do not put them in an `amsd` block. Other Spectre-language commands (such as your transient analysis command) must not appear in an `amsd` block.

If you pass a file containing an `amsd` block to `xrun` or to `xmelab`, you do not need to pass an analog simulation control file to `xrun` (or to `xmsim`), but the file must also contain your transient analysis statement in this case.

✔ If you have a `prop.cfg` file from a previous release, you can set the `AMSCB` environment variable as follows to enable the internal translator to convert your `prop.cfg` file to an `amsd` block file, `prop.cfg.scs`:

```
setenv MSCB YES
```

The first time you run the elaborator on the design, the translator converts your file, you verify the content, then run the elaborator again, this time, using the AMS control file instead of the old `prop.cfg` file.

Related Topics

- [Appendix E: Migration of `prop.cfg` file to an `amsd` Block Skip to end of metadata](#)
- [amsd Block Statements and Syntax](#)

amsd Block Statements and Syntax

Only the simulation front end (SFE) parser recognizes and processes the AMS statements that you can use in an `amsd` block. You can use AMS statements only in an `amsd` block, which has the following form:

```
amsd{  
    AMS_statements  
}
```

You may not reference global parameters in `parameter=value` assignments in AMS statements.

In `amsd` block statements, you must escape any characters that are not legal characters in the Spectre namespace using a backslash. For example: `busdelim=\+`

The following AMS statements can appear in an `amsd` block:

Statement	Description
<code>portmap</code>	The <code>portmap</code> statement tells the AMS Designer simulator how a SPICE subcircuit interface should appear to the elaborator
<code>config</code>	The <code>config</code> statement specifies which definition you want to use for particular cells or instances.

ie	The <code>ie</code> statement specifies interface element parameters, optionally for a particular design unit. If you do not specify a design unit, the elaborator applies the parameter settings to all interface elements globally.
ce	The <code>ce</code> statement specifies conversion element parameters for VHDL-to-SPICE connections, optionally for a particular design architecture (cell level), or for a specific pin (port level). If you do not specify a cell or a port name, the elaborator applies the parameter settings to all conversion elements globally.
connectmap	The <code>connectmap</code> card defines a resolution function for one or more disciplines by using the <code>realresolve</code> parameter. The syntax of the <code>connectmap</code> statement is the following:

For example:

```
* AMS control file -- amsdControl.scs
include "../source/design.scs" //analog netlist
include "../models/model.scs" //device model file
include "../models/diode.scs" section=dio
// diode.scs is the model file; "dio" is the section to use
include "../models/pmos1.scs" section=nom
// pmos1.scs is the model file; "nom" is the section to use
include "../analogControl.scs"
//analog control file

//amsd block
amsd{
    portmap subckt=pll_top autobus=yes
    config cell=pll_top use=spice
    ie vsup=2.0
}
```

You can put a single `amsd` block in a separate file. You can also have one or more `amsd` blocks in one or more Spectre or SPICE input files. For example:

```
// file1.scs
amsd{
    portmap subckt=s1 autobus=yes
    config cell=s1 use=spice
}

amsd{
    portmap subckt=s2 autobus=yes
    config cell=s2 use=spice
}

// file2.scs

amsd{
    portmap subckt=subcx autobus=yes
    config cell=subcx use=spice
}
```

Related Topics

- [portmap](#)
- [config](#)
- [ie](#)
- [ce](#)
- [connectmap](#)

portmap

The `portmap` statement tells the AMS Designer simulator how a SPICE subcircuit interface should appear to the elaborator.

```
portmap subckt= name [ parameter = value ]
portmap module= name [ parameter = value ]
portmap entity= name [ parameter = value ]
```

The `portmap` statement must always be placed before the `config` statement in an `amsd` block.

Valid `parameter=value` assignments for the `portmap` statement are as follows:

subckt	Name of the SPICE subcircuit or Verilog-A module to which to apply these <code>portmap</code> settings. Use this form (typically, together with <code>config use=spice</code>) when your design contains SPICE-on-leaf constructs. Valid Values: Any valid subcircuit name or a Verilog-A module. Here is an example on specifying a Verilog-A module using the <code>subckt</code> parameter. <pre>include "basicMos.scs" include "acf.scs" ahdl_include "inv.va" amsd{ portmap subckt=inv autobus=no config inst=top.DUT0.I0 use=spice ie vsup=1.8 rhi=5 rlo=5 tr=100p }</pre> Note: The asterisk wildcard (*) is supported at the end of the string while specifying the subcircuit name.	
module	Name of the Verilog module to which to apply these <code>portmap</code> settings. Use this specifier (typically, together with <code>config use=hdl</code>) when your design contains hdl components instantiated in SPICE blocks. Valid Values: Any valid module name.	
entity	Name of the VHDL entity to which to apply these <code>portmap</code> settings. Use this specifier (typically, together with <code>config use=hdl</code>) when your design contains hdl components instantiated in SPICE blocks. Valid Values: Any valid entity name.	
autobus	Indicates whether to bind Verilog buses to SPICE ports. Valid Values:	
	yes	Bind Verilog buses to SPICE ports.
	no	Do not bind Verilog buses to SPICE ports.
	Default: <code>yes</code>	
excludebus	List of entities not to map because they are not buses. No default.	
reversebus	Specifies the <code>reversebus</code> name. No default.	
busdelim	List of bus delimiters.Valid Values: <code>[]</code> , <code>_</code> , <code><></code> , <code>none</code> , or any single character Default: <code>[] <></code>	
casemap	Specifies casing for name mapping.Valid Values:	
	upper	Map all names to uppercase.
	lower	Map all names to lowercase.
	keep	Maintain name casing as it is.
	Default: <code>lower</code>	
portcase	Specifies casing for port names. Valid Values:	
	upper	Map all port names to uppercase.
	lower	Map all port names to lowercase.
	keep	Maintain port casing as it is.
	Default: <code>keep</code>	
file	Name of the port-bind file containing customized port bindings.	

reflib	Name of the library of precompiled cell containing custom port bindings. If the <code>reflib</code> parameter is provided with the <code>portmap</code> statement, the <code>reffile</code> parameter is not compiled even if it is specified with the <code>portmap</code> statement. The library of precompiled cell provided through the <code>reflib</code> parameter will be used instead of the <code>worklib</code> library. The use of <code>reflib</code> parameter eliminates the need for special compile from AMSCB. It also limits the precompiled cell search to a specific library, thereby improving performance. Note: The <code>reflib</code> parameter works only for VHDL-SPICE, and not for Verilog-SPICE. In the example given below, the software will search for the precompiled cell <code>s11</code> in the library <code>mylib</code>. Notice that the <code>reffile</code> parameter is not specified with the <code>portmap</code> statement. <pre>portmap subckt=s11 refformat=vhdl refflib=mylib config cell=s11 use=spice</pre>						
reffile	Name of the HDL reference file containing custom port bindings. If the reference file is not in the current working directory, include the path to its name, such as <pre>portmap ... reffile=/user/project/myfile.v ...</pre> The <code>reffile</code> parameter also supports UNIX environment variables. You can specify the file path using the UNIX environment variable <code>\$WDIR</code>, as follows: <pre>portmap ... reffile="\$WDIR/source/analog/PLL.v"</pre> Note: The path specified using the UNIX environment variable must be enclosed within double quotation marks. • Use <code>refformat</code> to specify the format of this file. (The default format is <code>verilog</code>.) • Use <code>porttype</code> to specify how to match the ports. (The default is by order.) The software uses this parameter as the HDL reference when automatically generating a port-bind file containing port bindings at SPICE-to-HDL boundaries. The reference file must satisfy the following requirements: • The reference file must compile standalone. • If the reference file contains any library declarations or package references, they must be available and compile successfully. • The reference file must not contain any external language constructs. Note: You do not need to specify the <code>reffile</code> parameter with the <code>portmap</code> card if you have specified it with the <code>xrun</code> command, unless you want to use a different <code>reffile</code> than what you specified to <code>xrun</code>.						
porttype	Specifies how to match ports. You must also specify a reference file. Note: The software uses this parameter when automatically generating a port-bind file. The port-bind file specifies custom bindings for the interface between SPICE and Verilog. Valid values: <table border="1"> <tr> <td>name</td><td>Match ports by name.</td></tr> <tr> <td>order</td><td>Match ports by order.</td></tr> <tr> <td colspan="2">Default: <code>order</code></td></tr> </table>	name	Match ports by name.	order	Match ports by order.	Default: <code>order</code>	
name	Match ports by name.						
order	Match ports by order.						
Default: <code>order</code>							
refformat	Specifies the format of the <code>reffile</code>. Valid Values: <table border="1"> <tr> <td>verilog</td><td>Verilog format See also "Binding Ports using a Verilog File".</td></tr> <tr> <td>vhdl</td><td>VHDL format</td></tr> <tr> <td colspan="2">Default: <code>verilog</code></td></tr> </table>	verilog	Verilog format See also "Binding Ports using a Verilog File".	vhdl	VHDL format	Default: <code>verilog</code>	
verilog	Verilog format See also "Binding Ports using a Verilog File".						
vhdl	VHDL format						
Default: <code>verilog</code>							
stub	Name of the cell for which you want to use the stub version; you must also specify a <code>match</code> assignment.						
match	Specifies which definition to use when matching the interface. The <code>match</code> assignment applies only when you also specify <code>stub</code> on the <code>portmap</code> statement and <code>use=stub</code> on the <code>config</code> statement. Valid values: <table border="1"> <tr> <td>verilog</td><td>Use the Verilog interface</td></tr> </table>	verilog	Use the Verilog interface				
verilog	Use the Verilog interface						

	spice	Use the SPICE interface
	Default: verilog	
input	Specifies a net as an input port. No default.	
output	Specifies a net as an output port. No default.	
inout	Specifies a net as an input and output port. No default.	
ignore	Ignores the specified ports. For example: <pre>amsd{ ie vsup=1 portmap subckt=invx2 porttype=name ignore="vdd vss" config cell=invx2 use=spice }</pre>	

Related Topics

- [Implicit and Explicit Port Mapping](#)
- [Spice-to-Verilog Port Mapping](#)

Implicit and Explicit Port Mapping

Depending on the design structure and your port mapping requirements, you can apply port mapping inside an amsd block in two ways:

- **Implicit portmap:** For SPICE-in-Middle and SPICE-on-Leaf design flows in Verilog-AMS, port mapping can be achieved without using the `portmap` statement. Only the `config` statement would suffice. The tool automatically generates implicit portmaps, with default values, for the subcircuits or modules referred to in the `configcard`. In the autogenerated portmap statement, the cell name used in the `config` statement is used as the SPICE subcircuit name or the Verilog module name, and the default values are assumed for all other parameters of the `portmap` statement.

Examples

- If the `config` statement is `config cell=divider use=hdl`, the following `portmap` statement would be autogenerated.

```
portmap module=divider
```
- If the `config` statement is `config cell=analog_top_a use=spice`, the following `portmap` statement would be autogenerated.

```
portmap subckt=analog_top_a
```

The tool assumes the following `portmap` parameter default values for autogenerating skeletons, commfile, or the portbind file: `autobus=yes`

```
busdelim="[] <>"
reformat=verilog
interconnect=mixed
```

To use a non-default value for any of the above parameters or to use `reffi` or `file` parameter with the `portmap` card, use an explicit portmap.

- **Explicit portmap:** In most situations, `portmap` and `config` statements are required in pairs to specify cell bindings. One pair of `portmap` and `config` statements can bind multiple instances of the same cell to either a SPICE port in a Verilog block or a Verilog port in a SPICE block.

Example

```
portmap subckt=foo autobus=yes portcase=lower busdelim=_ reformat=verilog
config inst="top.inst1 top.inst2 top.inst3" use=spice
```

Here, `inst1`, `inst2`, and `inst3` are instances of `foo` in the Verilog module `top`, which needs to be configured to SPICE.

Related Topics

- [portmap](#)
- [Spice-to-Verilog Port Mapping](#)
- [SPICE](#)

Spice-to-Verilog Port Mapping

In the AMS Designer simulator, if you have a SPICE instance in a Verilog module, you can specify the port mappings for the SPICE instance in any of the following ways:

- Using the reference file name with the `xrun` command
- Using a simple `portmap` card
- Using `reffile` in a `portmap` card with default of `porttype=order`
- Using `reffile` in a `portmap` card with default of `porttype=name`
- Using a file in a `portmap` card

In addition to the port mapping information between SPICE and Verilog representations of the given block, the automatically generated port bind file provides some more information in the form of comments. This information includes information like the SPICE subcircuit name and the Verilog module name for which the port bind file is generated.

Following is an example of a port bind file with some typical comments.

```
/** dummy_spice.pb **  
  
/* This file is automatically generated.  
  
/* (c) Copyright 2007-2008 Cadence Design Systems, Inc.  
  
/* All rights reserved.  
  
/*  
  
/* Portbind file for:  
  
/* SPICE subckt dummy_spice : HDL module dummy_spice  
  
/* SPICE port :      HDL port :      HDL Parameters  
  
  
{ a_0, a_1, a_2 }      :      a[0:2] :      dir=output vh_type=STD_LOGIC  
  
                                vh_basetype=STD_ULONGIC  
  
                                vh_converter=E2STD_LOGIC  
  
                                vh_vectortype=STD_LOGIC_VECTOR
```

Examples

In the following example, the `analog_top.cir` file contains the subcircuit definition for `ANALOG_top`. The `portmap` statement tells the elaborator not to map Verilog buses to SPICE ports (`autobus=no`) and not to change the case mappings between Verilog-AMS instantiations and SPICE subcircuits (`casemap=keep`). The `config` statement tells the elaborator to bind the `ANALOG_top` cell as a SPICE subcircuit; the `analog_top.cir` file contains the master.

```
include "analog_top.cir" // SPICE or Spectre format include file  
amsd{  
    portmap subckt=ANALOG_top autobus=no casemap=keep  
    config cell=ANALOG_top use=spice  
}
```

The following example shows how you can specify a custom port-bind file for a Verilog-SPICE boundary:

```
amsd{
  portmap subckt=analog_top file=top.pb
  config cell=analog_top use=spice
}
```

The following example shows how you can specify a custom port-bind file for a SPICE-in-the-middle design unit:

```
amsd{
  portmap module=nand2 file=nand2.pb
  config inst=top.a1.x1 use=hdl
}
```

The following example illustrates using a reference file for a SPICE-in-the-middle construct. Note that the default value of `porttype` (for the reference file) is `order`:

```
amsd{
  portmap subckt=analog_spice reffile=analog_spice.v autobus=yes
  config cell=analog_spice use=spice

  portmap module=nand2
  config cell=nand2 use=hdl
}
```

The following example shows how you might specify customized port bindings for a SPICE-on-leaf construct using a reference file:

```
amsd{
  portmap subckt=analog_spice reffile=analog_spice.v autobus=yes
  config cell=analog_spice use=spice
}
```

Here are some more examples:

```
amsd{
  portmap subckt=ana_gate reffile=ana_gate.v refformat=verilog
  config cell=ana_gate use=spice

  portmap module=nand2 file=nand2.pb
  config cell=nand2 use=hdl
}

amsd{
  portmap subckt=p11_top busdelim=_ file=p11_top.pb
  config cell=p11_top use=spice

  portmap module=divider file=divider.pb
  config cell=divider use=hdl

  portmap module=counter file=counter.pb
  config cell=counter use=hdl
}
```

SPICE Port-to-Verilog Bus Mapping Example

Verilog in file `top.v`:

```
module p11_top(refclk, p0_clk, reset);
  wire [1:0] p0_clk;
  ...
endmodule

module top;
  ...
  p11_top p1(...);
  ...
endmodule
```


Related Topics

- [portmap](#)
- [Implicit and Explicit Port Mapping](#)
- [SPICE-in-the-Middle](#)
- [SPICE](#)

SPICE

```
.subckt pll_top refclk reset p0_clk_0 p0_clk_1
...
```

There are five ways to specify the port mapping for the SPICE instance `p1` in Verilog module `top` :

- Using the reference file name with the `xrun` command:

```
xrun work.scs top.v

//work.scs
amsd{
...
portmap
subckt=top
config cell=top use=spice
}
```

With this specification, the port association between Verilog and SPICE will be established using the information from the Verilog reference file `top.v` specified in the `xrun` command. The port `p11_clk` in the resulting port bind file will be mapped as:

```
{ p0_clk_0, p0_clk_1 } : P0_CLK[1:0]
```

- Using simple portmap card:

```
amsd{
...
portmap subckt=pll_top
...
}
```

With this specification, the port association between Verilog and SPICE will be established using the information in the SPICE subcircuit. The port `p11_clk` in the resulting port bind file will be mapped as:

```
{ p0_clk_0, p0_clk_1 } : P0_CLK[0:1]
```

- Using `reffile` in portmap card with default of `porttype=order`

```
amsd{
...
portmap subckt=pll_top reffile="top.v"
...
}
```

With this specification, the port association between Verilog and SPICE will be established using the information from both the reference Verilog module and the reference SPICE subcircuit. In other words, their port order will be preserved from their original source files. The port `p11_clk` in the resulting port bind file will be mapped as:

```
{ p0_clk_0, p0_clk_1 } : P0_CLK[1:0]
```

- Using `reffile` in portmap card with `porttype=name`

```
amsd{
...
portmap subckt=pll_top reffile="top.v" porttype=name
...
}
```

With this specification, the port association between Verilog and SPICE will be established using the information in the reference Verilog module, both for the name and its port order. The port `p11_clk` in the resulting port bind file will be mapped as:

```
{ p0_clk_1, p0_clk_0 } : P0_CLK[1:0]
```

- Using `file` in portmap card

```
amsd{
...
portmap subckt=p11_top file="p11_top.pb"
...
}
```

With this specification, the port association between Verilog and SPICE will be established using the information in the given port bind file, `p11_top.pb`. You can specify the port binding information in the `p11_top.pb` file for port `p11_clk` to reflect the exact connection expected at the Verilog/SPICE boundary. For example:

```
{ p0_clk_1, p0_clk_0 } : P0_CLK[1:0]
```

Related Topics

- [portmap](#)
- [Spice-to-Verilog Port Mapping](#)
- [Implicit and Explicit Port Mapping](#)

config

The `config` statement specifies which definition you want to use for particular cells or instances.

```
config use=definition [parameter=value]
```

The `config` statement must always be placed after the `portmap` statement in an `amsd` block.

Valid `parameter=value` assignments for the `config` statement are as follows:

cell	One or more names of cell references in the NC HDLs for which you want to use the specified definition (<code>use=definition</code>). If you specify more than one cell name, you must use double quotation marks around the list: <pre>config cell="subA subB subC" use=spice config cell=dsp use=hdl</pre> Note: The asterisk wildcard (*) is supported at the end of the string while specifying the cell name.

inst	Full hierarchical path to one or more names of instance references in the NC HDLs for which you want to use the specified definition. If you specify more than one instance, you must use double quotation marks around the list. <pre>config inst="top.inst1 top.inst2 top.inst3" use=spice</pre> In addition, if the hierarchical path of an instance contains an index (vector bit), you must use double quotation marks around the instance definition. <pre>config inst="top.inv_chain.forgentblk[0].case_blk0.inv" use=spice</pre> You can use the <code>inst</code> assignment to specify iterated instances, use wildcards and escaped identifiers as shown in the examples below. Moreover, it is also possible bind multiple iterated instances from different hierarchies. For example: <pre>amsd{ portmap module=buf1 porttype=name autobus=yes config inst="tb.wrapper.dut.I0<0>.B<*>" use=hdl portmap module=buf1 porttype=name autobus=yes config inst="tb.wrapper.dut.I1<*>.B<0>" use=hdl portmap module=buf1 porttype=name autobus=yes config inst="tb.wrapper.dut.I2<*>.B<*>" use=hdl }</pre> You can specify multiple iterated instances with a single <code>inst</code> assignment as follows: <pre>portmap module=buf1 porttype=name autobus=yes reffile="buf.v" config inst="tb.wrapper.dut.I0<1>.B0<0> tb.wrapper.dut.I0<1>.B0<1> tb.wrapper.dut.I0<1>.B0<2>" use=hdl.</pre>	
use	Specifies which definition you want the simulator to use. Valid values:	
	hdl	Use the HDL definition
	spice	Use the SPICE definition
	stub	Replace the specified cell with a stub version (which has the same interface but no content); you must specify a <code>cell</code> parameter assignment; you can specify a <code>match</code> choice of <code>verilog</code> or <code>spice</code> in the <code>portmap</code> statement for the stub
No default. You must specify a <code>use</code> assignment.		

<code>exclude</code>	Specifies a scope that will be excluded from the application of the <code>use</code> parameter setting in the <code>config</code> card. In the example below, the <code>use=hdl</code> setting will be applied to all instances of cell <code>inv</code> except those within the scope <code>mid2_block</code>, which is a block within <code>mid_block</code>. <pre>amsd{ ... config cell=mid_block use=spice config cell=inv use=hdl exclude=mid2_block }</pre>

You must specify either a cell or an instance or both. If you specify both, the instance must be an instance of the cell.

Here are some examples:

```
amsd{
  config cell="subA subB subC" use=spice
  config inst=top.I1.I2 use=spice
  config cell=dac use=hdl
}

include "analog_top.cir"
amsd{
  portmap stub=analog_top match=spice // SPICE stub
  config inst=top.xana_top2 use=stub

  portmap subckt=analog_top autobus=yes // SPICE subcircuit
  config inst=top.xana_top3 use=spice

  portmap subckt module=nand2 file=nand2.pb // SPICE-in-the-middle
  config inst=top.a1.x1 use=hdl

  portmap stub="worklib.analog_top:module" match=verilog // Verilog stub
  config inst=top.xana_top4 use=stub
}
```

Related Topics

- [Verilog 2001 Configuration File](#)

Verilog 2001 Configuration File

The `amsd` block supports Verilog 2001 configuration (specified using the `-libmap` and `-compcnfg` `xrun` options). If the configuration information is specified in the `amsd` block as well as the Verilog 2001 configuration file, then the following rules apply:

- Configuration information specified in the `amsd` block takes higher priority than Verilog 2001 configuration.
- For instances that are not specified as `use=spice` in the `amsd` block, Verilog 2001 configuration is considered.

For Verilog 2001 configuration, the top-level design unit is specified using the design parameter in the configuration file.

Consider an example where you have the following:

<pre>//file top.v module top(); clock a1(...); // clock a2(...); clock a3(...); endmodule</pre>	<pre>//file clock.v - > module clock(...); //rtl description endmodule</pre>	<pre>//clock.vg -> gateLib clock(...); //gate-level description endmodule</pre>

The Verilog 2001 configuration file config01.v is specified as follows:

```
config cfg1;
design rtlLib.top
default liblist rtlLib;
instance top.a3 liblist gateLib:
endconfig
```

In a pure digital run, the binding result would be as follows:

```
a1; a2=>rtlLib; a3=>gateLib
```

If you want to configure instance `a1` as spice view, you can use the `config` statement in the `amsd` block as follows:

```
include "clock.scs"
amsd{
  portmap subckt=clock
  config inst= top.a1 use=spice
}
```

When the simulation is run, the binding result would be as follows:

```
a1=>spice ; a2=>rtlLib ; a3=>gateLib
```

For more information on how to configure the contents of a design, refer to *Section 13* of the *Verilog 2005 LRM*.

Related Topics

- [config](#)

ie

The `ie` statement specifies interface element parameters, optionally for a particular design unit. If you do not specify a design unit, the elaborator applies the parameter settings to all interface elements globally.

You can use an `ie` statement to automate the process of creating a custom discipline and connect rule for connecting the custom discipline to the `electrical` discipline. The software applies the custom discipline to domainless nets in your design. So, if you have digital modules with undeclared port disciplines, you can use an `ie` statement to specify a discrete discipline for domainless nets and the elaborator will insert the appropriate connect module automatically.

If you have digital or Verilog-AMS modules with ports of discipline `logic`, you must use `-amsconnrules` to specify the connect rule name to use for connecting `logic` ports to `electrical` ports. You can use both mechanisms (`ie` statements and `-amsconnrules`) at the same time for designs that contain both digital modules with undeclared port disciplines and digital or Verilog-AMS modules with `logic` ports.

The syntax for an `ie` statement is as follows:

```
ie vsup=supplyValue | connrules=inhconn_full [scope=scopeValue] [parameter=value] | currentmode
```

It is mandatory to specify either `vsup=supplyValue` or `connrules=inhconn_full` in the `ie` statement. `vsup=supplyValue` specifies the positive supply value in reference to `vss` (voltage of source and substrate, or the voltage level of the negative supply of IE). By default, `vss` is 0, while it can be specified as any non-zero value in the `ie` statement.

The `currentmode` parameter enables current mode processing for connect modules. The value can be 1 or 0. By default, the value is 0.

Using ECL logic, you may need to use negative supplies values for interface elements. For example, your logic might operate between - 5V to -3.2V. In this case, for both `vsup` and `vss`, you will specify the `ie` statement as:

```
ie vsup=1.8 vss=-5
```

The `vsup` value is defined as `vdd-vss`. Therefore, here, `vsup = (-3.2 - (-5)) = 1.8`, which is a positive value.

For AMS `ie` card setup, the `vsup` value needs to be positive because it is a relative value compared to `vss` to ensure that `vhi > vlo`.

`connrules=inhconn_full` uses inherited connection properties for power supply and ground nets for connect modules.

For System Verilog User Defined Nettype (UDN) to Electrical connections, if you are using custom UDN and connect module, you must specify the `nettype` parameter in the `ie` card statement. The following example illustrates custom connect module insertion for a custom nettype `myNT`:

```
ie vsup=5.0 nettype="myNT"
```

During elaboration time, these rules are used to select the appropriate connect modules and insert them into the design.

If you are using the built-in UDN, you do not have to explicitly specify the `nettype` parameter in the `ie` card statement. The tool automatically identifies and inserts the built-in UDN, "EEnet".

The `ie` statement supports the connect rule parameters specified in the following built-in connect rules that are provided in the IUS installation hierarchy:

- `CR_full_fast`
- `CR_full`
- `CR_basic`
- `CR_inhconn_full_fast`
- `CR_ss_full_fast`
- `CR_inhconn_full`
- `CR_ss_full`
- `CR_dynsup_full_fast`

These built-in connect rules are regular connect rules with certain design style so that the all connect rule parameters are supported in `ie` statement transparently as `ie` parameters.

The AMS control block supports the `ie` connect rules and parameters solely based on the built-in connect rule definition. The connect rules name can be either the full name (`CR_full`) or the sub name (`full`). For example:

```
ie connrules=full vsup=1.8 rout=1000
```

```
ie connrules=CR_full vsup=1.8 rout=1000
```

You can specify zero or more scopes and zero or more parameter assignments (customizations). Any customization you specify apply to domainless nets only.

The software automatically builds the "full-fast" connect rule with the voltage supply level you specify (unless you use the `connrules=connRules` parameter assignment). You do not need to use `-amsconnrules`; you do not need to specify the connect module path or to compile any connect modules.

The software also applies any `parameter=value` customizations you specify, automatically, and writes information to the `xrun.log` file.

Related Topics

- [Dynamic Voltage Supply](#)
- [Custom Connect Rules](#)
- [Scope Assignments](#)
- [Parameter Assignments](#)

Dynamic Voltage Supply

The `ie` statement also supports dynamic voltage supply. It supports the connect rule parameters of the built-in `CR_dynsup_full_fast` connect rule that you can use to specify the dynamic voltage supply. For example:

```
ie [connrules=CR_dynsup_full_fast] vddnet=<value> vssnet=<value> [other parameters]
```

The software automatically builds the `dynsup_full_fast` connect rule with the dynamic voltage supply that you specify.

Parameter `connrules` is optional. However, it is mandatory to specify both `vddnet` and `vssnet` parameters to point to the dynamic power supply net and ground net in the design hierarchy. For example:

```
ie connrules=CR_dynsup_full_fast //illegal; parameters vddnet and vssnet are not specified

ie vddnet="top.e1" vssnet="top.e0" //legal

ie connrules=CR_dynsup_full_fast vddnet="top.e1" // illegal; parameter vssnet is not specified

ie connrules=CR_dynsup_full_fast vssnet="top.e0" // illegal; parameter vddnet is not specified
```

You can also specify hierarchical dynamic voltage supply for Verilog-AMS electrical and wreal nodes/nets. Hierarchical dynamic voltage supply provides similar functionality like the supply-sensitive connect modules where a block uses its own voltage supply for the connect modules.

SPICE nodes are not supported with hierarchical dynamic voltage supply.

Hierarchical dynamic voltage supply only supports a single identifier (for example, `vddnet=vdd1`) rather than hierarchical paths (for example, `top.vdd1`), along with appropriate cell or inst scope assignments. For example,

```
amsd{

ie vddnet= "vdd1" vssnet="vss1" cell="foo"

}
```

In this example, `vdd1` and `vss1` can be the ports of cell `foo`, as specified by the `cell` scope assignment. This means that all DVS connect modules inserted within the scope of any instance of `foo` will have their `vddnet/vssnet` set to `vdd1` and `vss1` respectively. Refer to the Examples section below for a detailed example on hierarchical dynamic voltage supply.

Hierarchical dynamic voltage supply is enabled by using the `xrun` option `-hier_dvs` and can work together with normal dynamic voltage supply specification.

Parameters `vddnet`, `vssnet`, `cell`, and `inst` also support Linux environment variables as values. For example, you can specify the following Linux environment variables:

```
setenv vdd_cell1 tb.avdd
setenv vss_cell1 tb.avss
setenv vdd_cell2 tb.vddc
setenv vss_cell2 tb.dvss
setenv CELL1 cell1_wrap
setenv CELL2 cell2_wrap
setenv INST1 tb.dut.I0
```

You can reuse the already defined environment variables in the AMSD control file, `amsd` block, and `ie` card statements using the `$` character followed by the environment variable name. For example:

```
ie vddnet=$vdd_cell1 vssnet=$vss_cell1 cell=$CELL1
ie vddnet=$vdd_cell2 vssnet=$vss_cell2 cell=$CELL2
ie vddnet=tb.avdd vssnet=tb.avss instport=duv.tb.oomr_observe_net inst=$INST1
```

Related Topics

- [ie](#)
- [Custom Connect Rules](#)
- [Scope Assignments](#)
- [Parameter Assignments](#)

Custom Connect Rules

In addition to the built-in connect rules, the `ie` statement also supports custom connect rules (user-defined connect rules). You can specify your custom connect rule file using the `xrun` command and use the `connrules` parameter to specify the connect rule that you want to generate based on the specified connect rule file. For example:

```
xrun top.vams amscf.scs myCR.vams

//amscf.scs

amsd{

ie vsup=1.8
```

```
ie vsup=5 connrules=myConnrule inst=top.al vdelta=0.1n
ie vsup=3.3 connrules=myConnrule cell=middle nparam=0.5
```

Here, two connect rules will be generated based on `myConnrule`. Note that the parameters specified above, such as `vdelta`, `nparam`, and `vsup` should be text macros specified in the `myCR.vams` connect rule file. In addition, parameters configured through the `ie` statement should be valid `ie` parameters that are text macros specified using the ``define` directive in the `myCR.vams` connect rule file. For example:

```
`define supply 1.8                      //Supply voltage
`define tran 0.2n                       //Transition rising time
`define RO 300                          //Output resistance
`define V_delta (`supply)/64.0          //Voltage delta of absdelta
`define V_tol_Delta (`V_delta)/4.0      //Voltage tolerance of absdelta
`define T_tol_Delta (`tran)/20.0
```

You can also specify a custom rule in the `ie` statement with only the scope setting and without reconfiguring any parameter, as shown below.

```
//amscf.scs
amsd{
...
ie connrules myConnrule inst=top.al
...
}
```

You can import the user-defined connect rule file using the following methods:

- Adding the file directly at the `xrun` command line.
`xrun ... myCR.vams`
- Compiling to a specific library using the `xrun -makelib` option.
`xrun ...-makelib amslib myCR.vams -endlib`
- Using the precompiled library directory with the `xrun -reflib` option.
`xrun .. -reflib myConn_lib`

`xrun` reads the custom connect rule file and generates a new set of internal disciplines and connect rules based on the specified `ie` parameters. Therefore, the user-defined connect rule file supported by AMS should contain the text macro definitions of all parameters used in the `ie` statement and appropriate connect statements. ``ifdef` is not supported in the user-defined connect rule file.

Related Topics

- [ie](#)
- [Scope Assignments](#)
- [Parameter Assignments](#)

Scope Assignments

If you do not specify a scope assignment, the scope is global. Valid `scope=scopeValue` assignments are as follows:

inst	Full hierarchical path to an instance to which you want to apply the specified interface element parameters. If you specify more than one instance, you must separate each instance with a space and enclose the string in double quotation marks. The <code>inst</code> assignment also supports iterated instances, wildcards, and escaped identifiers. For example: <pre>ie vsup=5 inst="tb.wrapper.dut.I0<0>.B0<0>" ie vsup=3.3 inst="tb.wrapper.dut.\I0<1>.B0*"</pre> You can also specify multiple hierarchies with a single <code>ie</code> statement with a space between them, as shown below. <pre>ie vsup=5 inst="tb.wrapper.dut.I0<0>.B0<0> tb.wrapper.dut.I0<1>.B0<1>"</pre>
cell	Specification of a cell to which you want to apply the specified interface element parameters. If you specify more than one cell, you must separate each cell with a space and enclose the string in double quotation marks.
instport	Full hierarchical path to one or more names of instance ports to which you want to apply the specified interface element parameters. If you specify more than one name, you must separate each name with a space and enclose the string in double quotation marks. For example: <pre>ie vsup=1.2 instport="a b c"</pre> Note: The functionality of <code>instport</code> assignment has been merged into <code>net</code>, for the same scope. Therefore, if you specify both the assignments, the <code>net</code> assignment takes higher precedence. However, if the <code>net</code> assignment is not specified, the <code>instport</code> assignment is considered.
cellport	One or more cell port names to which you want to apply the specified interface element parameters. If you specify more than one cell port, you must separate each cell port name with a space and enclose the string in double quotation marks.

net	One or more net names to which you want to apply the specified interface element parameters. If you specify more than one name, you must separate each name with a space and enclose the string in double quotation marks. The <code>net</code> assignment also supports the specification of instance ports. As a result, the functionality of <code>instport</code> parameter has been merged into <code>net</code> for the same scope. If you specify both <code>instport</code> and <code>net</code> parameters in the <code>ie</code> statement for the same scope, the <code>net</code> assignment will have higher precedence. For example: <pre>amsd{ ie vsup=4 net=top.clk ie vsup=1.8 instport=top.clk }</pre> In the above example, <code>top.clk</code> specified with the <code>net</code> assignment will have higher priority. The <code>net</code> assignment also supports iterated instances and escaped identifiers. For example: <pre>ie vsup=2 net="top.I1.X1<1>.I2.net1" ie vsup=3.2 net="tb.wrapper.\I0<*> .B1.x1"</pre>
cellupport	One or more cell port names. The software applies the interface element parameters to upper-level connections (ports or nets) to the specified cell port or ports. If you specify more than one name, you must separate each name with a space and enclose the string in double quotation marks.
lib	Logical name for library of design units to which you want to apply the specified interface element parameters.

The scope-based `ie` card setting, by default, applies the `-setdiscipline` option to force the specified scope level with a discrete logic discipline and overrides the previous default and detailed discipline resolution process.

If you have a design that uses connect modules across discrete nets (such as R2L), it is recommended to pair `cellport` with `cellupport` for cell ports definition and `instport` with `net` for instance port definition.

```
amsd{
    ie vsup=1.8 //Global ie for full design
    ie vsup=3.3 vthi=1.8 vtlo=1.5 cellport="cell_A.*" cellupport="cell_A.*"
        // cell_A ports will be using different discipline than global
    ie vsup=5.0 net=top.net1 instport= top.inst_1.net1
// Instance inst_1 port net1 which is connected to top.net1 will use vsup=5.0
}
```

The reason for this use model is `cellport` and `instport` will define the internal ports while `cellupport` and `net` will define the connecting outside ports; they need to match in discipline.

You can use the wildcard character(s) in the scope name to specify more than one scope at a time. For example:

```
ie vsup=1.2 cellport="top.*_vdddig"
```

could match `top.jbb_vdddig_1`, `top.jbb_vdddig_2`, `top.jbc_vdddig_1`, `top.jbc_vdddig_2`, and so on.

For hierarchical-related specifications, that is, `inst`, `instport`, and `net`, wild cards do not match across the hierarchical levels unless the wildcard is specified at the end of the string. For example, `"a.*.d"` would match `a.b.d`, but not `"a.b.c.d"`. However, `"a.*"` could match `"a.b"` and `"a.b.c.d"`.

Related Topics

- [ie](#)
- [Dynamic Voltage Supply](#)
- [Custom Connect Rules](#)
- [Parameter Assignments](#)

Parameter Assignments

The software uses any parameter assignments you specify to customize the connect rules. Any customizations you specify apply to domainless nets only. Valid `parameter=value` assignments for the `ie` statement are as follows:

<code>vthi</code>	Voltage value above which the simulator assigns a logical <code>1</code> . The simulator determines the default value from the connect rule.
<code>vtlo</code>	Voltage value below which the simulator assigns a logical <code>0</code> . The simulator determines the default value from the connect rule.
<code>vx</code>	Final real number for logical <code>x</code> . The simulator determines the default value from the connect rule.
<code>tr</code>	Rise time for analog transition, from <code>vtlo</code> to <code>vthi</code> or <code>vx</code> . Default value: 0.2 ns
<code>rlo</code>	Output resistance for L2E when digital input is <code>0</code> . Default value: 200 Ohms
<code>rhi</code>	Output resistance for L2E when digital input is <code>1</code> . Default value: 200 Ohms
<code>rx</code>	Output resistance for L2E when digital input is <code>x</code> . Default value: 40 Ohms
<code>rz</code>	Output resistance for L2E when digital input is <code>z</code> . Default value: 10M Ohms
<code>txdel</code>	Controls the amount of wait time before a digital port is driven to <code>x</code> for the connect modules <code>E2L</code> , <code>E2L_2</code> , <code>L2E</code> , <code>Bidir</code> , and <code>Bidir_2</code> . Measured in nano seconds. Default value: Four times the <code>tr</code> parameter. If the <code>tr</code> parameter is not specified for the <code>ie</code> statement, the simulator uses the default <code>tr</code> value 0.2n and calculates the default <code>txdel</code> value as 0.8n. Example: <pre>amsd{ ie vsup=1.8 txdel=0.9n }</pre>
<code>mode</code>	Specifies the type of connect module insertion in the <code>ie</code> card. You can assign two values to this parameter: • <code>merged</code> - This is the default value. The merged value instructs the elaborator to insert a single merged connectmodule for all the nets that are connected to the same port and require the same connectmodule. • <code>split</code> - The <code>split</code> value instructs the elaborator to insert a separate connectmodule for every net connected to the same port, irrespective of whether or not they require the same connectmodule.
<code>vpso</code>	Converts the PSO X state to a user-supplied voltage in AMS-CPF.

vdelta	Voltage delta value ranging from 0 to vsup. Default value: <code>vsup/64</code>	
vtol	Voltage tolerance value ranging from 0 to vdelta. Default value: <code>vdelta/4</code>	
vtrlo	Ratio of low threshold during value conversion. Default value: <code>vtlo/vsup</code> .	
vtrhi	Ratio of high threshold during value conversion. Default value: <code>vtrhi/vsup</code> .	
vrtol	Ratio of tolerance during value conversion. The default value is <code>vtlo/vsup</code> .	
currentmode	Enables wreal current mode processing. Default value: 0	
connrules	Connect rule to build using the <code>vsup=supplyValue</code> you specify. Valid values:	
	basic	Build the "basic" connect rule
	full	Build the "full" connect rule
	inhconn_full	Build inheritance-based connect rule. The connect rule defined using the <code>inhconn_full</code> argument can inherit supply voltage automatically. The default supply is <code>cds_globals.\vdd!</code> and <code>cds_globals.\vss!</code> You can use this argument instead of using the <code>vsup</code> parameter to specify the supply voltage. However, if both <code>vsup</code> and <code>connrules=inhconn_full</code> exist in the same <code>ie</code> card, an error is reported. For example: <code>ie vsup=1.8 connrules = inhconn_full</code>
	full_fast	Build the "full-fast" connect rule
	Default Value: <code>full_fast</code> Note: You can find the set of connect rule files that Cadence provides--including the " <code>basic</code> ", " <code>full</code> ", " <code>inhconnfull</code> ", and " <code>full-fast</code> " connect rules--in <code>your_install_dir /tools/affirma_ams/etc/connect_lib</code> . See <code>your_install_dir /tools/affirma_ams/etc/connect_lib/README</code> for detailed information about them.	
discipline	Used as the discipline name corresponding to the <code>ie</code> card. If this parameter is not specified, the discipline name is auto-generated. In the example below, the discipline name corresponding to the <code>ie</code> card in the first <code>ie</code> statement will be <code>logic1_8</code> instead of the auto-generated discipline name <code>ddiscrete_1_8</code>. However, because the <code>discipline</code> parameter is not used with the second and third <code>ie</code> statements, the corresponding discipline names for these <code>ie</code> cards will be <code>ddiscrete_3_3</code> and <code>ddiscrete_4_5</code>. <pre>amsd{ ie vsup=1.8 discipline=logic1_8 rx=25 ie vsup=3.3 rlo=125 ie vsup=4.5 tr=0.4n }</pre> Important: The <code>discipline</code> parameter can take only discrete discipline values. In the example below, the discipline value is not discrete and therefore it is not supported. <pre>ie vsup=1.8 instport="top.sub.pin" discipline="electrical"</pre> For continuous disciplines, use the elaboration <code>-setdiscipline</code> option instead. For example: <pre>-setdiscipline "instterm-top.sub.pin- electrical"</pre> Note: Unlike in a connectmap card, the <code>discipline</code> parameter in an <code>ie</code> card can accept only a single discipline value.	

<code>enable_highz</code>	If set to <code>1</code>, converts HighZ state reported by the Spectre <code>dyn_highz</code> check at the analog (SPICE) port to logic Z at digital (Verilog) port of AMS connect modules. This parameter is supported only by the AMSD built-in default connect rules (<code>full_fast</code>) on electrical-to-logic and bidirectional conversion. Possible values are <code>0</code> and <code>1</code>. The default value is <code>0</code>. Note: Conversion of HighZ state results in delay of one transient step. Example: <pre>ie vsup=1.8 enable_highz=1 net="top.n1 top.n5"</pre> To enable the <code>ie enable_highz</code> feature, the AMSD built-in Verilog-AMS connect modules are enhanced with system function <code>\$dyn_highz</code>, which takes two arguments. The first argument is the analog port of the Verilog-AMS module and the second is to enable the highz conversion. It returns a boolean/integer value, where <code>1</code> means high-Z and <code>0</code> means non-high-Z. Example: <pre>highzState = \$dyn_highz(Ain, enable_highz);</pre>
<code>nox</code>	If set to <code>1</code>, disables converting to logic X when the electrical signal is in between <code>vthi</code> and <code>vtlo</code> in electrical to logic conversion, for the purpose of simpler and faster verification. Default value: <code>0</code>

Examples

In the following example, all digital nets that connect to analog in the `top.I3` scope have interface elements with `vsup=4.5` and `tr=1.2`; all digital nets that connect to analog in the scope of instance `top.I1` of cell `mid1` have interface elements with `vsup=1.8`; all other nets use the global value, `vsup=5.0`:

```
amsd{
  ...
  ie vsup=5.0
  ie inst=top.I3 vsup=4.5 tr=1.2n
  ie vsup=1.8 cell=mid1 inst=top.I1
}
```

Here is an example showing how you can specify more than one instance (scope) to use the same supply voltage:

```
amsd{
  ...
  ie vsup=4.5 inst="testbench.vlog_buf"
  ie vsup=4.5 inst="testbench.vlog_buf1"
  ie vsup=4.5 inst="testbench.vlog_buf2"
}
```

You can also specify more than one instance in a single statement like this:

```
amsd{
  ...
  ie vsup=4.5 inst="testbench.vlog_buf testbench.vlog_buf1 testbench.vlog_buf2"
}
```

Here is an example showing how you can specify different instances (scopes) to use different supply voltages:

```
amsd{
  ...
  ie vsup=1.8 inst="testbench.vlog_buf"
  ie vsup=3.0 inst="testbench.vlog_buf1"
  ie vsup=4.5 inst="testbench.vlog_buf2"
}
```

The following examples showing how you can specify a supply voltage you want to apply to an instance port, a cell port, a net, or to all domainless nets connected to a cell port:

```
amsd{
  ...
  ie vsup=1.8 instport="top.I1.in" tr = 0.4n
}

amsd{
  ...
  ie vsup=1.8 cellport="mid1.w" vtlo=0.7 vthi=1.5
}

amsd{
```

```
...
ie vsup=1.8 net="top.n1" rlo=150 rhi=240
}

amsd{
...
ie vsup=1.8 cellupport="mid2.w" rx=25
}
```

Here is an example showing how you can specify more than one scope on a single `ie` statement:

```
amsd{
...
ie vsup=1.8 cell="mid" inst="top.I1" cellupport="mid.w"
}
```

Here is an example showing how you can build a set of "full" connect rules using a 1.8 Volt supply value:

```
amsd{
ie vsup=1.8 connrules="full"
}
```

In the following example, the program applies the same custom discipline to cell `mid1` and to instance `top.I1` (because the parameter = value assignments in the `ie` statements are exactly the same):

```
amsd{
ie vsup=1.8 cell="mid1" tr=0.3n rlo=200
ie vsup=1.8 inst="top.I1" tr=0.3n rlo=200
}
```

The example below describes a global `ie` card with the default `merged` connect module insertion. In addition, it describes a scoped `ie` card for the `divider` block, which is parameterized for a `split` connect module insertion.

```
ie vsup=1.8
ie vsup=3.2 mode=split cell=divider
```

The example below describes how the new `vpso` parameter can be used with the `ie` card to convert the PSO X state to a user-supplied voltage in AMS-CPF.

```
amsd{
ie vsup=3.3 instport="testbench.vlog_buf.I1.in" vpso=0.1
}
```

The example below describes how to specify the hierarchical dynamic voltage supply.

```
module top;

electrical e1, e2, e3;

electrical a11, a10, a21, a20, a3;

foo f1 (e1, a11, a10);
foo f2 (e2, a21, a20);
foo f3 (e3, a3, a3);

initial begin // simulation outputs
#10 $display(V(e1),,V(e2),,V(e3));
#20 $display(V(e1),,V(e2),,V(e3));
end

analog begin // setup supplies
V(a11) <+ 1.8;
V(a10) <+ 0.18;
V(a21) <+ 3.3;
V(a20) <+ 0.33;

end

endmodule

module foo(p, a1, a0);
```

```
output p, a1, a0;

electrical a1, a0; // local supplies

assign #20 p = 1;

endmodule

//amsd block

ie vddnet=a1 vssnet=a0 inst="top.f1 top.f2" // hierarchical DVS

ie vddnet=top.a21 vssnet=top.a10 inst=top.f3 // normal DVS

//command line

xrun top.vams sim.scs -ieinfo -hier_dvs
```

Note that hierarchical dynamic voltage supply works for a single identifier (not OOMR style) used in `vddnet/vssnet`, like `a1/a0` in the above example.

When the simulation is run, the tool will insert three IEs:

```
top.e3__L2E_2_dynsup__ddiscrete_dynsup_full_fast_ie2 for the second IE card, as normal DVS
top.e2__L2E_2_dynsup__ddiscrete_dynsup_full_fast for the first IE card (Hierarchical DVS)
top.e1__L2E_2_dynsup__ddiscrete_dynsup_full_fast for the first IE card
```

The simulation result would be as follows:

```
0.18 0.33 0.18
1.8 3.3 3.3
```

Related Topics

- [ie](#)
- [Dynamic Voltage Supply](#)
- [Custom Connect Rules](#)
- [Scope Assignments](#)

ce

The `ce` statement specifies conversion element parameters for VHDL-to-SPICE connections, optionally for a particular design architecture (cell level), or for a specific pin (port level). If you do not specify a cell or a port name, the elaborator applies the parameter settings to all conversion elements globally.

Using a `ce` statement, you can define and select VHDL conversion elements for VHDL-to-SPICE connections in your design. You can specify the values of various generics in a conversion element to control the accuracy and performance of the inter-kernel value conversions. Optionally, you can define a conversion element for a given scope of the design by specifying a cell name. Alternatively, you can specify a port name to define a conversion element for the specific pin. If you do not specify a cell or a port, the `ce` statement applies to the entire design.

The syntax for a `ce` statement is as follows:

```
ce name=ceName type=sigType [dir=portDir] [cell=archName] [genericmap=value] [cellport=portName] [excludeport=portName]
[priority=high|low] [optimize=on|off]
```

Valid parameter assignments for the `ce` statement are as follows:

name	Conversion element name using one of the following forms: <pre>library. cell library. cell :arch</pre> For example: <pre>name=mylib.std_logic2e name=mylib.std_logic2e:arch</pre>
type	Digital signal type that connects to SPICE; for example: <pre>type=std_logic</pre>

dir	(Optional) Direction of the digital port Valid values:	
	in or input	Input port
	out or output	Output port
	inout	Bidirectional port
	Default value: inout	
cell	(Optional) Name of the cell defining the scope of the <code>ce</code> statement; for example: <code>cell=dummy_spice</code>	
genericmap	Parameter for overriding generic values in the conversion element; for example: <code>genericmap="vsup 3.3 vthi 2.2"</code>	
cellport	(Optional) Name of the port (or ports) to which the elaborator will apply the parameter settings specified in the <code>ce</code> statement; for example: <code>cellport="vdd2 gnd2"</code>	
excludeport	(Optional) Name of the port (or ports) that will not use the parameter settings specified in the <code>ce</code> statement; for example: <code>excludeport="p0 nx"</code>	
priority	(Optional) Priority of the conversion element for optimization. This value determines the conversion element to be used for optimization, when there are multiple conversion elements of different signal types. For example: <pre>ce ... type=base priority=low ce ... type=derived priority=high</pre> In the above example, if the conversion element optimization process has to choose one of the conversion elements for optimization, it will choose the derived conversion element because it has <code>high</code> priority.	
optimize	(Optional) The global switch to turn off conversion element optimization. Default is <code>on</code>. The following statement for the global <code>ce</code> card implies that conversion element optimization will be ignored for the entire design. <pre>ce optimize=off</pre> The following statement, on the other hand, implies that optimization will be ignored for scope <code>nand2</code> and below only. Conversion element optimization will occur for all other scopes. <pre>ce optimize=off cell=nand2 ...</pre>	

For example:

```
ce name=mylib.std_logic2e genericmap="vsup 2.4 vthi 1.8 vtlo 1.2" cell=dummy_spice type=std_logic dir=input
```

Here is an example in the context of an `amsd` block:

```
amsd{
  portmap subckt=mult16x16_spice autobus=yes refformat=vhdl
    reffile=mult16x16_spice.vhd
  config cell=mult16x16_spice use=spice
  ce name=my_ad_lib.std_logic2e dir=input type=std_logic
  ce name=my_ad_lib.e2std_logic dir=out type=std_logic
}
```

connectmap

The `connectmap` card defines a resolution function for one or more disciplines by using the `realresolve` parameter. The syntax of the `connectmap` statement is the following:

```
connectmap [discipline=value(s)] realresolve=resolveFunction
```

where:

- The `discipline` parameter specifies the names of discrete disciplines separated by spaces. You can specify one or more discipline names with this parameter.

- The `realresolve` parameter specifies the name of a predefined `wreal` resolution function to be associated with the specified disciplines. This is a mandatory parameter that can have any of these values:

- `default`
- `fourstate`
- `sum`
- `avg`
- `min`
- `max`
- `none`

If you specify the value of the `realresolve` parameter as `none`, the disciplines listed in the `connectmap` card will use the global resolution function. An error will occur if you do not specify the `realresolve` parameter or specify an invalid value for this parameter.

In the following example, the first `connectmap` card will display an error because no `realresolve` parameter is defined, while the second `connectmap` card will set the `disc3` and `disc4` disciplines to use the `max` resolution function for `wreal` nets. The third and the last `connectmap` card in the example will cause the `disc5` discipline to use the global resolution function for the purpose of resolving the resolution function of a connection.

```
amsd{  
  
    connectmap discipline="disc1 disc2"  
  
    connectmap discipline="disc3 disc4" realresolve=max  
  
    connectmap discipline="disc5" realresolve=none  
  
}
```

Integration between connectmap and ie Cards

You can use both the custom and auto-generated disciplines of the `ie` card in the `connectmap` card as follows:

```
amsd{  
    ie vsup=1.8 discipline=logicl_8 rx=25  
    ie vsup=3.3 rlo=125  
    ie vsup=4.5 tr=0.4n  
    connectmap resolution=max discipline="logicl_8 ddiscrete_4_5"  
}
```

The first `ie` statement in the above example explicitly associates the `logicl_8` discipline with the `ie` card. The next two `ie` statements do not use the discipline parameter, and therefore, have the auto-generated discipline names `ddiscrete_3_3` and `ddiscrete_4_5`. The `connectmap` statement defines the `max` resolution function for both the custom `logicl_8` discipline and the auto-generated `ddiscrete_4_5` discipline.

In addition to the explicitly defined resolution functions, the global and default resolution functions apply to `ie` custom and auto-generated disciplines as well.

Related Topics

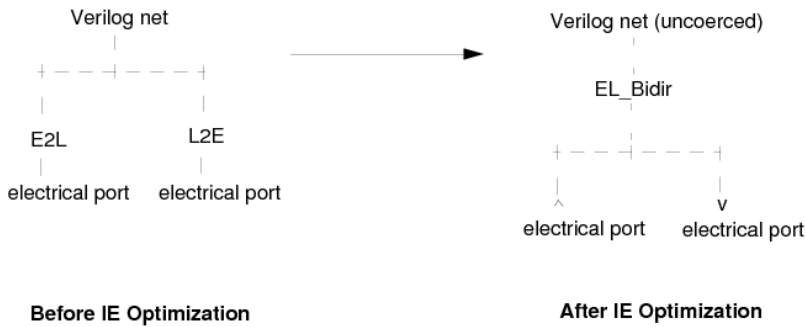
[connectmap](#)

Hierarchical Interface Element Optimization

Single-Level IE Optimization

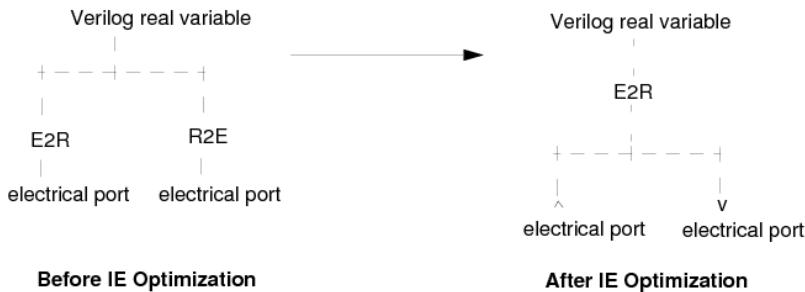
Single-level IE Optimization is a process in which the elaborator inserts a single bidirectional interface element (IE) in place of multiple IEs when the following conditions are met:

- A verilog net (logic, real, or user-defined type) forced to digital (using `$monitor`, `$display`, `force`, OOMR reference in digital behavioral code, and so on) is connected to more than one analog port at the **same hierarchical level**.
- Multiple dissimilar IEs exist between the digital net and the analog ports.



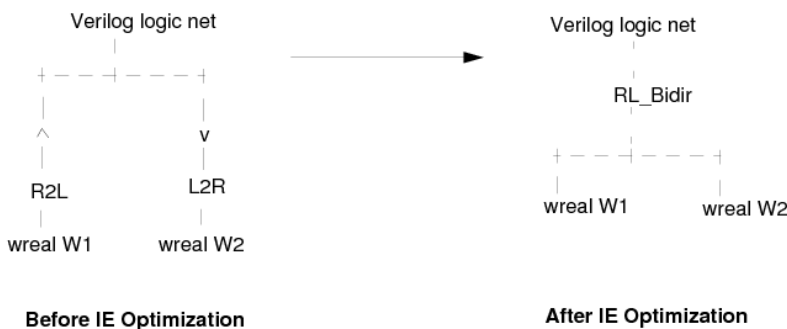
In case of analog/digital variable connections, multiple dissimilar IEs are replaced with a single electrical-to-real (E2R) IE that is connected to all the analog ports when:

- A real variable is connected to more than one analog port at the **same hierarchical level**.
- Multiple dissimilar IEs exist between the variable and the analog ports.



In case of nets involving real/logic connections, multiple dissimilar IEs between the logic net and the real or user-defined ports are replaced with a single real-to-logic (R2L) bidirectional IE when:

- A verilog net forced to digital (using `$monitor`, `$display`, `force`, OOMR reference in digital behavioral code, and so on) is connected to more than one wreal or user-defined type port at the same hierarchical level.
- Multiple dissimilar IEs exist between the logic net and the real or user-defined ports.



Hierarchical Optimization

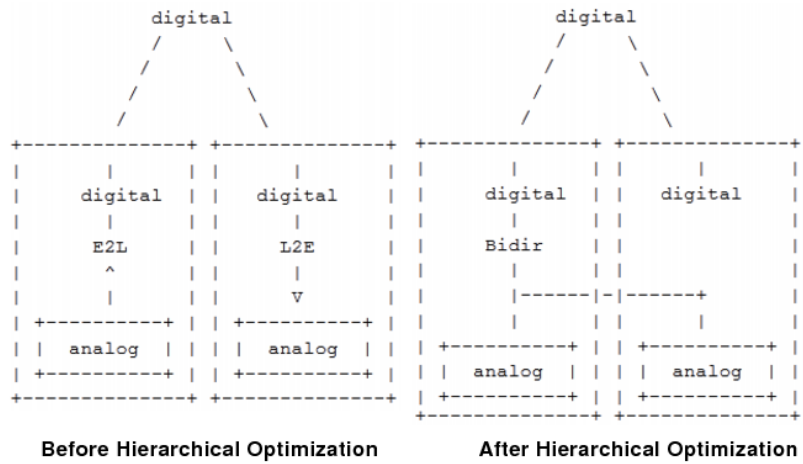
The interface elements (IEs) that are instantiated at any point on a hierarchical digital net can be combined into one. As a result, all IE-connected nets are collapsed while still maintaining the connection between the analog and digital nets. For this, the following criteria must be met:

- For discrete (logic or real) to electrical connections, there must be an electrical port to which all the IEs and electrical nets can be collapsed.
- For real net to logic connections, real nets are collapsed. However, logic nets are not collapsed.
- Multiple R2L or R2E IEs connected to a network of connected real variables and multiple connected real variables in different hierarchical scopes are

not collapsed.

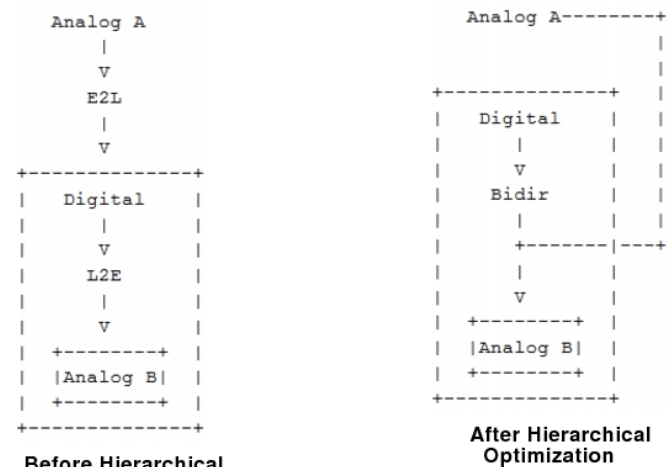
Examples

Digital Over Electrical Connections



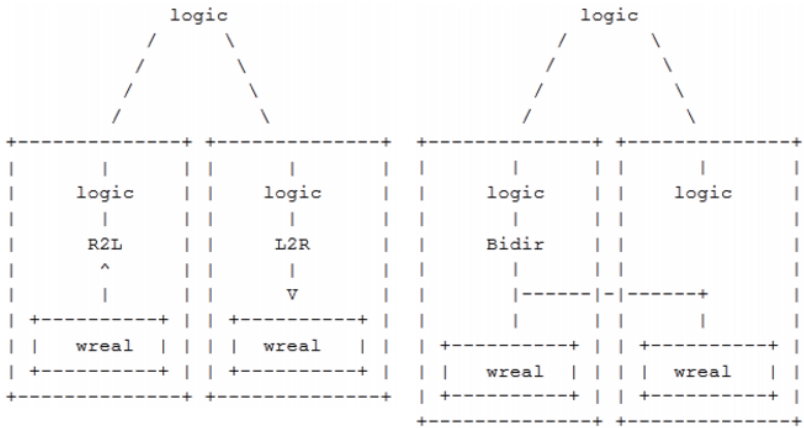
One analog net is connected to the other port, and both IEs are replaced with a single bidirectional IE.

Digital Sandwich



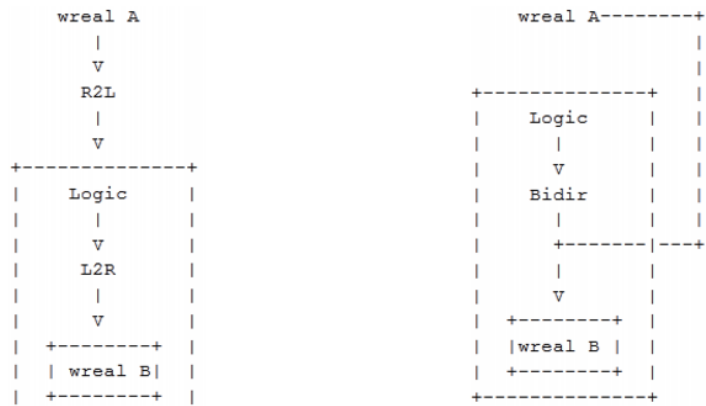
The uppermost analog net is collapsed into the bottom port and both IEs are replaced with a single bidirectional IE.

Logic over Wreal



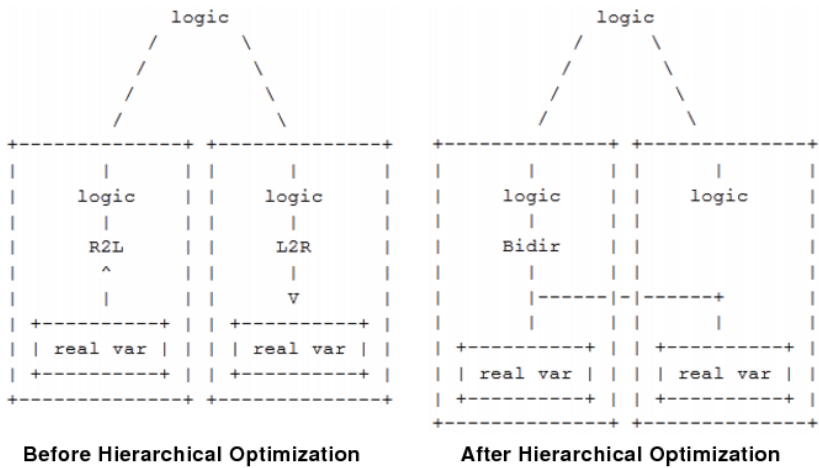
Before Hierarchical Optimization **After Hierarchical Optimization**
Both IEs are replaced with a single bidirectional IE and one wreal net is connected to the other port.

Logic Sandwich



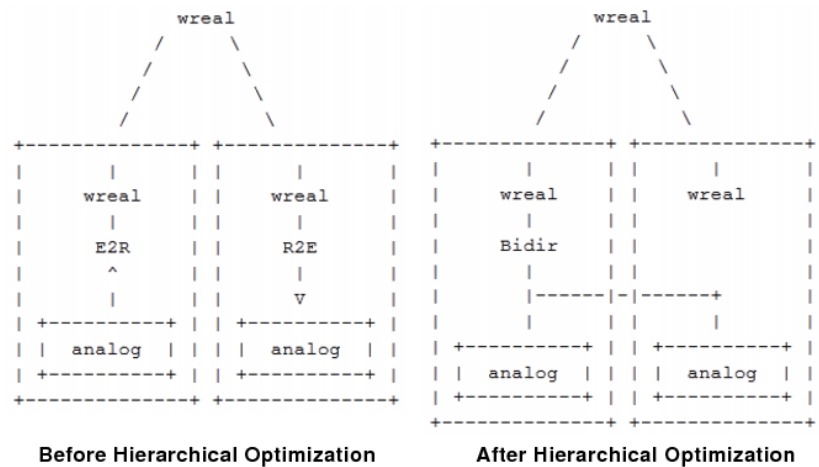
Before Hierarchical Optimization **After Hierarchical Optimization**
The uppermost wreal is collapsed into the bottom port and both IEs are replaced with a single bidirectional IE.

Logic over Real Variable



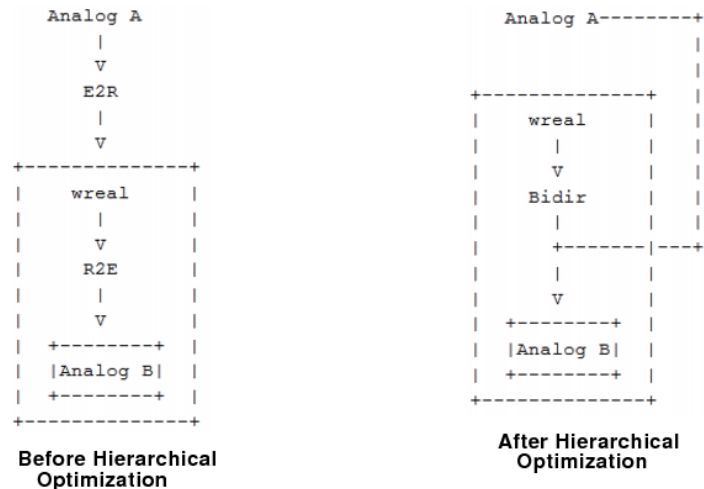
The two IEs are replaced with a single bidirectional IE and both real variables are connected to it.

Wreal over Electrical



One analog net is connected to the other port and both IEs are replaced with a single bidirectional IE.

Wreal Electrical Sandwich



The uppermost analog net is collapsed into the bottom port and both IEs are replaced with a single bidirectional IE.

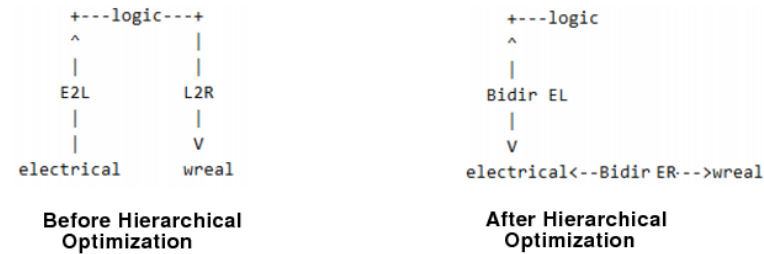
Interface Element Optimization between Dissimilar Nets

In Interface Element (IE) optimization between dissimilar nets, the elaborator inserts a bidirectional IE to collapse the electrical signal onto the real just like two electrical signals are collapsed in normal IE optimization. The structure of the resultant optimization depends on the following:

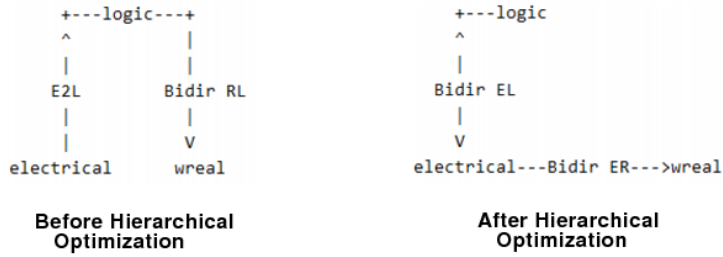
- The type of connect modules in the original configuration
- The type of objects (variables or nets) connected

Examples

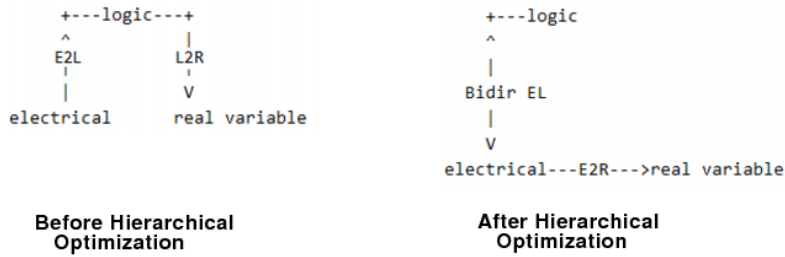
Simple E-L-R connection



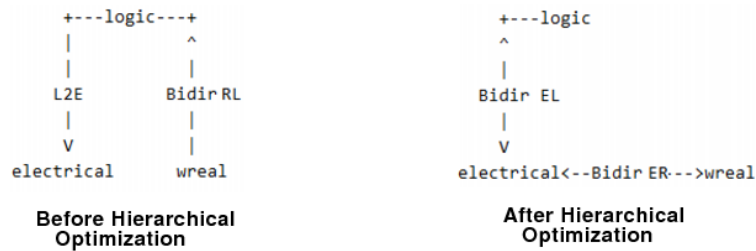
E-L-R with Bidirectional RL Connection



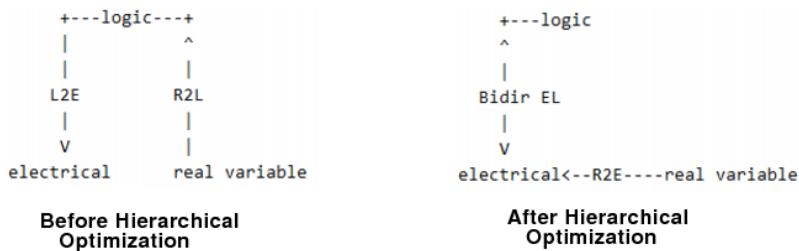
E-L-Real Variable



R-L-E with Bidirectional Connection

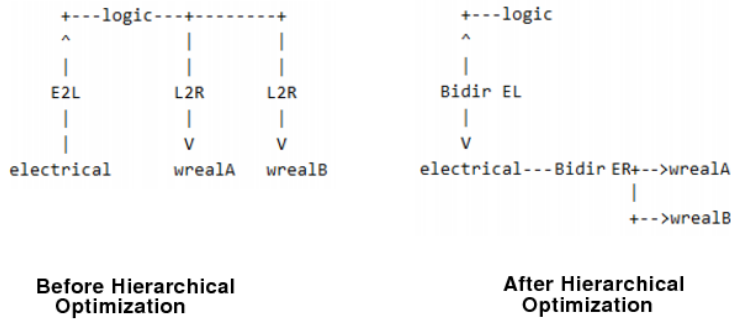


Real Variable-L-E Connection



Multiple Interface Elements

Multiple interface elements of the same type will be collapsed together during optimization, as shown below.



Hierarchical IE optimization is enabled by default. Hierarchical IE optimization is supported in the AMS-CPF flow. Currently, IE optimization is applied on nets only if the drivers/loads of the nets are associated with the same power domain.

According to the purpose of accuracy improvement, IE optimization does not occur for the following cases:

- Electrical net over logic ports
- Real variable over logic ports
- Real variable over electrical ports

You can use the `-ieinfo xmelab/xrun` option to generate a report that contains the connect module information used in the design.

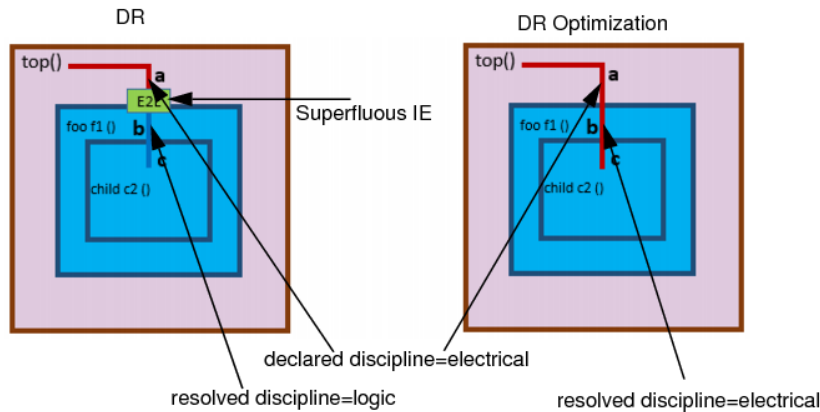
Related Topics

- [Discipline Resolution Optimization](#)
- [Xcelium Mixed Signal: Co-Simulation Functionalities](#)

Discipline Resolution Optimization

In Spectre AMS Designer, insertion of interface elements (IEs) is based on discipline resolution (DR). IEs are inserted along the analog/digital boundary, if a net is set, declared, or resolved to a digital discipline and connects to an analog node. Nets with declared discrete discipline express an explicit digital domain, whereas, nets with resolved or set discipline are interconnects that acquire their domain during elaboration.

Sometimes, interconnects are not used in digital context, however, they still acquire a discrete discipline, which causes superfluous IEs to be inserted along the analog/digital boundary. The simulator automatically handles such situations and optimizes these nets to an analog node. As a result, an interface element is not required for such connections.



```
module top ()
  electrical a;
  foo f1 (a);
  analog begin
    v(a) <+3;
  end
endmodule

module foo (b);
  input b;
  child c2 (b);
endmodule

module child (c);
```

Due to DR, nets `top.f1.b` and `top.f1.c2.c` could be set to digital discipline and the tool inserts E2L for the connection.

Since both nets have no read/write access in the design, the E2L connection can be optimized.

DR optimization works only on implicit (set/resolved) discrete nets including wreal nets (declared or coerced). It does not work for L2R, LRE, and VHDL-Verilog connections. It also does not work for explicit (declared) discrete net and SV-DC nettypes.

Discipline resolution optimization can be enabled by using the `xrun` option `-amsdropt`.

Unlike in a connectmap card, the `discipline` parameter in an `ie` card can accept only a single discipline value.

Related Topics

- [connectmap](#)
- [Hierarchical Interface Element Optimization](#)
- [Xcelium Mixed Signal: Co-Simulation Functionalities](#)

Xcelium Mixed Signal: Co-Simulation Functionalities

As part of the application-centric enhancements in the Xcelium mixed-signal verification, a set of related SystemVerilog and SPICE co-simulation use cases can be now enabled using a single `xrun/xmelab` command-line option, `-dms_cosim`.

If you are running co-simulations for your design or verification, you can use the `-dms_cosim` option to automatically enable the following technologies for a more optimized and out-of-box experience with mixed-signal designs:

- Automatically process Out-Of-Module References to SPICE (`-autospiroomr`)
- Optimize analog connectivity across digital/SV blocks using discipline resolution optimization (`-amsdropt`) while shutting off the more comprehensive discipline resolution algorithm from Verilog-AMS standard (`-disres none`)
- Enable advanced SystemVerilog and mixed-signal connectivity improvements such as connecting SystemVerilog Hierarchical UDT to electrical, SystemVerilog interface to DMS/Verilog-AMS signals, SystemVerilog multi-dimensional array to SPICE/electrical ports, coercion for SystemVerilog interfaces/program/checker blocks, using wildcard-named port connections in SV bind statements (all using one option `-sv_ms`).

The following is an example of a design where discipline resolution is enabled (`-disres none` is disabled) because the design has mixed signal content that requires more comprehensive mixed-signal elaboration.

```
command: top.sv -64 amscf.scs -timescale 1ns/100ps -dms_cosim child.vams
```

```

=====top.sv=====
module top;
wire [0:2]foo ;
spice_driver spice_drvr0 (foo); //SPICE DRVR
dig_receiver dig_rcvr0(foo);
endmodule

module dig_receiver(input wire [0:2]in);
spice_dut rcvr0 (in); //SPICE DUT
endmodule

module spice_driver (output wire [0:2] out); //DIGITAL VERSION OF spice_driver
endmodule

module spice_dut(input wire [0:2] in); //DIGITAL VERSION of spice_dut
endmodule

=====child.vams=====
`include "disciplines.vams"
module child();
electrical a;
endmodule

=====amscf.scs=====
include "./ana_gate.sp"
amsd {
portmap subckt=spice_driver autobus=yes file="./spice_driver.pb"
config cell=spice_driver use=spice
portmap subckt=spice_dut autobus=yes file="./spice_dut.pb"
config cell=spice_dut use=spice
ie vsup = 5.0
}

=====ana_gate.sp=====
simulator lang=spice
.subckt spice_dut in_0 in_1 in_2
.ends
.subckt spice_driver out[0] out[1] out[2]
.ends
=====

```

All such designs consisting of .vams, .vhams, or .svams, are considered as a co-simulation application and the following DISCOS warning is displayed:

```

xmclab: *W,DISCOS: [DMSINFO] '-disres none' which is one of the automated -dms_cosim optimizations is disabled for this design because the
design has mixed signal content which requires more comprehensive mixed-signal elaboration.

```

The following is an example of how using the `-dms_cosim` option optimizes mixed-signal elaboration algorithms for AMS co-simulation applications (SV+SPICE). Auto-detected co-sim design, switching elaboration to co-simulation(`-disres none` and `-amsdropt`) mode.

```

command: xrun top.sv -64 amscf.scs -timescale 1ns/100ps -dms_cosim
=====top.sv=====
module top;
wire [0:1][0:2]foo ;
spice_driver spice_drvr0 (foo[0]); //SPICE DRVR
spice_driver spice_drvr1 (foo[1]); //SPICE DRVR
dig_receiver dig_rcvr0(foo);
endmodule

module dig_receiver(input wire [0:1][0:2]in);
spice_dut rcvr0 (in[0]); //SPICE DUT
spice_dut rcvr1 (in[1]); //SPICE DUT
endmodule

module spice_driver (output wire [0:2] out); //DIGITAL VERSION OF spice_driver
endmodule

module spice_dut(input wire [0:2] in); //DIGITAL VERSION of spice_dut
endmodule

=====amscf.scs=====
include "./ana_gate.sp"
amsd {
portmap subckt=spice_driver autobus=yes file="./spice_driver.pb"
config cell=spice_driver use=spice
portmap subckt=spice_dut autobus=yes file="./spice_dut.pb"
config cell=spice_dut use=spice
ie vsup = 5.0
}

```

```
=====ana_gate.sp=====
simulator lang=spice
.subckt spice_dut in_0 in_1 in_2
.ends
.subckt spice_driver out[0] out[1] out[2]
.ends
=====
```

Related Topics

- [Hierarchical Interface Element Optimization](#)
- [Discipline Resolution Optimization](#)

Universal Connect Module (UCM)

The Universal Connect Module (UCM) consolidates and replaces the existing connect modules in `connectLib` in the Xcelium installation. It is enabled by default in AMS Spectre FX mode. To enable UCM in APS and Spectre X mode, use the `xrun` option `-ams_uctm`.

The UCM supports the `basic`, `full_fast`, and `full` features in one set of connect modules with connect rule `uCR`:

For electrical-logic	For electrical-real	For logic-real
uE2L	uE2R	R2L
uL2E	uR2E	L2R
uBidir	uERBidir	RL_bidir

The UCM supports the following constant and dynamic power supply features in one set of connect modules.

- Constant supply (`vsup`)
- Dynamic supply (`vddnet`, `vssnet`)
- Inherited connection supply
- Supply sensitive supply
- LPS

Related Topics

- [-ams_uctm](#)
- [UCM Use Model](#)
- [UCM Compatibility with Legacy Connect Rules](#)

UCM Compatibility with Legacy Connect Rules

The following table shows the parameterized feature compatibility of UCM with the legacy connect rules `CR_full_fast`, `CR_full` and `CR_basic`:

IE Card Parameter	Default Value Compatibility						Functionality
	Current CR_full	accuracy=conservative	Current CR_full_fast	accuracy=moderate	Current CR_basic	accuracy=liberal	
tr	200p	200p	200p	200p	200p	200p	Rise/full time of transition between vhi and vlo
ignore_xz	0	0	0	0	1, fixed	1	Ignores X in L-to-E conversion

vx	$(vhi+vlo)/2$	$(vhi+vlo)/2$	vlo	vlo	IGNORED	vlo	The voltage of X in L-to-E conversion
vz	$(vhi+vlo)/2$	$(vhi+vlo)/2$	vlo	vlo	IGNORED	vlo	The voltage of Z in L-to-E conversion
vtol	vsup/30	vsup/30	vsup/12	vsup/12	vsup/6	vsup/12	Signal cross-tolerance in E-to-L conversion
rx	$(rlo+rhi)/10$	$(rlo+rhi)/10$	rout	rout	IGNORED	rout	Resistance of X in L-to-E conversion
analog_load	1	1	0	0	0, fixed	0	Feedback analog loading effect to the digital receiver in L-to-E
nox	0	0	0	0	1, fixed	1	Does not produce X in E-to-L conversion
ramp_type	1	1	0	0	0, fixed	0	Signal rise/fall transition ramp type in the electrical contribution of L-to-E

Related Topics

- [-ams_ucm](#)
- [Universal Connect Module \(UCM\)](#)
- [UCM Use Model](#)

UCM Use Model

UCM can be enabled by the `xrun` command option `-ams_ucm` for all IE card statements.

```
xrun -ams_ucm amxcf.scs
```

With UCM, you can use the IE card option `accuracy`.

```
ie accuracy=moderate | liberal | conservative
```

The `accuracy` option lets you set the connect module features compatible with the existing connect rules `CR_basic`, `CR_full_fast`, or `CR_full` respectively. The default value for this option is `moderate`. Any feature in the full set of features can be parameterized in an IE card:

```
ie accuracy=liberal vsup=1.2 nox=1 ignore_xz=0
```

To enable the dynamic power supply features in UCM, use the IE card parameter `supply`. If not specified, UCM supports constant power and LPS by default.

```
ie supply=SS | INH | DVS
```

By default, the parameter `supply` sets the accuracy command to `moderate`:

```
ie supply=DVS vddnet=top.I1.I2.vdd
```

Using the parameters `vddnet` or `vssnet` with the `accuracy` command enables `supply=DVS` by default.

Example:

```
ie accuracy=liberal vddnet=top.I1.I2.vdd
```

Related Topics

- [Universal Connect Module \(UCM\)](#)

Analog Primitives and Subcircuits

You can instantiate analog primitives and subcircuits in your design. The supported analog primitives for the Spectre AMS Designer Simulator include SPICE and Spectre subcircuits and Verilog-A modules that you bring into the design using `ahdl_include` statements in the Spectre language. The AMS Designer simulator also automatically recognizes and supports all the Spectre built-in primitives except

- `a2ao`
- `a2d`
- `d2a`
- `node`
- `prst`

In the AMS Designer simulator, the maximum number of ports that can be used for primitives with infinite ports, such as `pvcvs`, `pvccs`, and `nport`, is 30.

You can instantiate these analog primitives with ports that you bind either by name or by order. Ensure that you don't mix these two types of port bindings for a particular analog primitive instantiation and always bind the parameters of an instantiated analog primitive by name.

Determining the Discipline of Analog Primitive Ports

The elaborator assumes the discipline of an analog primitive port is always `electrical`, regardless of what you might actually have specified. This assumption affects the discipline resolution of nets attached to analog primitive ports.

The AMS Designer simulator does not support either port-discipline attributes or discipline resolution for analog primitives as described in the Verilog-AMS LRM 2.1.

Related Topics

- [Port Bindings](#)
- [Analog Instances Inside Generate Statements](#)

Port Bindings

You can connect an instance of an analog primitive by linking the formal port name of the Spectre or SPICE master with the actual name declared in the instantiating module. For example, in the following instantiations `1` and `2` are the formal port names for the Spectre built-in primitive `resistor` and `p` and `n` are formal port names for the Spectre subcircuit `my_subckt_cap`.

```
resistor #(.r(100)) R1 (.1(src), .2(out));  
my_subckt_cap #(.c(50n)) C1 (.p(out), .n(gnd));
```

The formal names of the ports for Spectre built-in primitives can be found in the "Component Statements" chapter of Spectre Circuit Simulator Reference manual.

The following capabilities apply to the port expressions of the instantiations of analog primitives when you are binding ports by name.

- Formal port names can appear in arbitrary order.

```
module top;  
    electrical n1,n2,n3;  
    mysub sub1(.p3(n1), .p1(n2), .p2(n3)); // Arbitrary order  
endmodule  
  
subckt mysub(p1 p2 p3)  
    ...  
ends mysub
```

- Master ports can be left unconnected. The unconnected ports are handled as unbound ports and are left as floating nodes.

```
module top;  
    electrical n1,n2,n3;  
    mysub sub1(.p2(n3)); // p1 and p3 are unconnected  
end  
  
module subckt mysub(p1 p2 p3)  
    ...  
ends mysub
```

- Port lists can be used for documentation only, if desired.

```
module top;
    electrical n1,n2,n3;
    mysub sub1(.p1(), .p2(), .p3()); // Used for documentation
endmodule

subckt mysub(p1 p2 p3)
    ...
ends mysub
```

- The formal name of a node is not allowed to appear more than once in an instantiation. If a formal name appears more than once, an error occurs during elaboration.

```
// Contains illegal code
module top;
    electrical n1,n2,n3;
    mysub sub1(.p1(n1), .p2(n2), .p2()); // This is an error
endmodule

subckt mysub(p1 p2 p3)
    ...
ends mysub
```

Binding Ports by Order

You can also connect an instance of an analog primitive by implicitly linking the formal port name of the Spectre or SPICE master with the actual name declared in the instantiating module. For example, the following instantiations omit the formal port names for the Spectre built-in primitive `resistor` and for the Spectre subcircuit `my_subckt_cap` but use the known ordering of those names to establish connections with the actual names used in the instantiating module.

```
resistor #(.r(100)) R1 (src, out);
my_subckt_cap #(.c(50n)) C1 (out, gnd);
```

Related Topics

- [Guidelines for Binding Ports by Name or Order](#)
- [Analog Primitives and Subcircuits](#)

Guidelines for Binding Ports by Name or Order

Be aware of the following guidelines for port expressions which apply whether you bind ports by name or order.

- When a port expression has a range, only single bit selections are supported. For example:

```
module top;
    electrical [0:2]n;

    mysub sub1(.p1(n[0]), .p2(n[1]), .p3(n[2])); // Single bit selections.
//mysub sub1( n[0], n[1], n[2] );           // By order.
endmodule

subckt mysub(p1 p2 p3)
    ...
ends mysub
```

- Out-of-module port expressions must be scalars.

```
module cds_globals ( );
    electrical \vss! ;
    electrical \gnd! ;
    ground \gnd! ;
endmodule

module top;
    electrical lgnd;
    ground lgnd;

    vsource #(.dc(5), .type("dc")) V0 (.p(cds_globals.\vss! )); // Scalars.
//vsource #(.dc(5), .type("dc")) V0 (cds_globals.\vss! ); // By order.

    resistor #(.r(5K)) r1 (cds_globals.\vss! , lgnd); // Scalars only.
//resistor #(.r(5K)) r1 (.1(cds_globals.\vss! ), .2(lgnd)); // By name.
endmodule
```

Related Topics

- [Port Bindings](#)

Parameter Binding

The parameters of analog and Spectre built-in primitives can be set only by name. For example, you can instantiate a resistor like this:

```
resistor #(.r(1K)) R1(pos,neg) ;
```

The formats used to set the parameters of some Spectre instantiations differ from the equivalent instantiations supported by the AMS Designer simulator. These differences are described in the following table:

Difference in Instantiation	Examples of Equivalent Spectre and Verilog-AMS Instantiations
Spectre-enumerated parameters are supported as strings in Verilog-AMS	<pre>// Spectre V1 p n vsource type=dc ddc=5 // Verilog-AMS vsource #(.type("dc"),.ddc(5))V1(p,n); // Spectre N0 (in1 gnd in2 gnd) nport file="spiral.spdat" interp=splinerelerr=0.01 abserr=5e-4 // Verilog-AMS nport #(.file("spiral.spdat"), .interp("spline"), .relerr(0.01),.abserr(5e-4)) N0 (.t1(in1), .b1(gnd), .t2(in2), .b2(gnd));</pre>
Automatically-sized parameter arrays are passed differently	<pre>// Spectre Filter in gnd out gnd svcvsv poles=[1 0 5 0] // Verilog-AMS svcvsv #(.poles({1,0,5,0})) Filter(in,ground,out,ground);</pre>

Related Topics

- [Analog Instances Inside Generate Statements](#)

Analog Instances Inside Generate Statements

The AMS Designer Simulator supports analog instances instantiated inside the generate block hierarchy. The following generate statements are supported.

- **generate-for** - Also called for-generate, it enables the AMS module items to be instantiated multiple times using a for-loop construct. For example:

```
module vs_vams ();  
  
... ..  
  
genvar j;  
  
generate  
    for (j=0; j<3; j=j+1) begin  
        //Analog instance  
        beh_vsource V1 (a[i],gnd);  
        //SPICE primitive  
        resistor #(.r(100k)) R1 (a[j], gnd);  
    end  
endgenerate  
endmodule
```

- **generate-if** - Also called if-generate, it enables the AMS module items to be conditionally instantiated, based on the evaluation of an if-else condition.

```
module vs_vams ()  
  
... ..  
  
generate  
    if ( behModel) begin:behmod  
        beh_vsource V1 (in,gnd);  
        beh_resistor R1 (in, gnd);  
    end  
    else begin: spiprim  
        vsource #(.dc(5)) V1(in,gnd)  
        vsource #(.r(100k)) R1(in,gnd)  
    end  
endgenerate  
endmodule
```

- **generate-case** - Also called case-generate, it enables the module items to be conditionally instantiated, based on a select one-on-many case construct.

```
module vs_vams ()
    ... ..
generate
    case: param
    0:
        begin:gen1
        beh_vsource V1 (in,gnd);
        beh_resistor R1 (in, gnd);
    end
    1:
        begin:gen1
        vsource #(.dc(5)) V1 (in,gnd);
        resistor #(.r(100k)) R1 (in,gnd);
        end
    endcase
endgenerate
endmodule
```

Currently, the following is not supported:

- Analog behavioral block inside the generate statement
- Declaration of local parameters and nets
- User-defined functions

Related Topics

- [Analog Primitives and Subcircuits](#)

Subcircuits and Models

To prepare to simulate using a SPICE or Spectre language model, you must give the elaborator the location of the model file. Two ways you can specify the location of model files are as follows:

- Define the `MODELPATH` variable in the `hdl.var`
- Use the `-modelpath` option for `xmelab`

To include a subcircuit in your design, use an `amsd` block and reference subcircuit files.

If you have an undefined node in a subcircuit, you must declare that node using an explicit `global` statement in the top-level scope of your design. For example, the following statement declares two global signals:

```
global 0 gnd vdd!
```

Related Topics

- [Analog Primitives and Subcircuits](#)

Compiled C Flow

The compiled C flow is turned on by default in AMS for the Spectre solver.

The AMS Designer simulator uses the compiled C flow for all analog and mixed-signal blocks implemented in Verilog-A or Verilog-AMS, including interface elements. The compiled C flow also supports VHDL-AMS modules for improved simulation performance.

The compiled C flow stores the compiled code and the shared objects in a new directory that gets created in the directory where the simulation runs.

The compiled C flow is enabled by default and cannot be disabled. As a result, the `CDS_AHDL_CMI_ENABLE` environment variable and the Spectre `-ahdlcom` option that were used to turn on/off the compiled C code flow have been removed.

Incremental Compilation of C Code

In normal Compiled-C flow, each behavioral module generates a C-code source file from bytecode. The C-code file is compiled and linked to a shared library (*.so). The shared library does not change if the behavioral module does not change between two runs.

The AMS Designer Simulator has been enhanced to reuse the existing shared library (*.so),

instead of creating a new one, if the design does not change between two simulation runs.

The simulator recompiles the code under the following condition:

1. `%xrun test1.scs test2.scs test.vams`
2. `%xrun test2.scs test1.scs test.vams`

In the example above, even though #2 is the exact rerun of #1 with the SPICE file names interchanged, the simulator will start recompiling the code. This is because the simulator always uses the file name of the first `.scs` file as the design name and treats the #2 run as a change in the design. Therefore, incremental compilation will not take effect.

Disabling the Compile C Flow

Set the following environment variable to disable the VHDL-AMS compiled C flow:

```
setenv CDS_VHDLA_CC_FLOW NO
```

When the compiled C flow is disabled, the Byte code technology is used.

Related Topics

- [Using the Compiled C Code Flow](#)

Multi-Threading

In the AMS Designer simulator, multi-threading is used to speed up the device evaluation and matrix solving part of computation that often dominates large and post-layout circuit simulation. Multi-threading enables the high performance simulation engine providing significant performance gain and high capacity SPICE simulation on single-core, multi-core, and multi-cpu shared memory systems. This enables you to quickly simulate large pre-layout and post-layout designs. This feature provides scalable multi-threading performance on up to 16 cores. Multithreading is supported when Spectre is chosen as the analog solver.

Using mtline in a Schematic with AMS Simulator

AMS Designer supports using multiconduction transmission line ([mtline](#)) components in your designs.

To prepare an `mtline` instance for use in a schematic, do the following:

1. In the Virtuoso Schematic Editing window, select the placed mtline symbol in the schematic.
2. Choose *Edit - Properties - Objects*.
The Edit Object Properties form appears.
3. Click the *Invoke 'LMG' parameter extraction tool* button.
The Transmission Line Model Generator window appears.
4. Enter the parameters needed to characterize the mtline instance.
5. In the Transmission Line Model Generator window, choose *File - LRCG file name*.
The LRCG File Name form appears.
6. In the *LRCG Name* field, specify a name for the data file to be created.
7. Click OK.
8. In the Transmission Line Model Generator window, click *Calculate Parameters*.
The calculated parameters appear in the top of the window.
9. (Optional) Choose *Options - Output file Control - RLCG file*.
This turns off the generation of a macromodel, which is not used in AMS Designer.
10. Click *Create Macromodel*.
The Macromodel Creation notice appears with a message similar to the following:

```
Per-unit-length RLCG file is generated and written to the file myw_line.dat
```
11. Click *OK*, in the Macromodel Creation notice.
12. (Optional) In the Transmission Line Model Generator window, choose *File - Quit*.
The window closes.
13. In the Virtuoso Schematic Editing window, select the placed mtline symbol in the schematic.
14. Choose *Edit - Properties - Objects*.
The Edit Object Properties form appears.
15. In the *RLCG data file* field, type the full absolute path to the data file that you specified in step 6.
16. Ensure that the *use lmg subckt* button is not checked.
17. Click *OK*.

At this point, the mtline instance is ready for netlisting.

Related Topic

- [Using mtline in a SPICE or Spectre Subcircuit or Model](#)

Using mtline in a SPICE or Spectre Subcircuit or Model

For this approach, you use LMG to create the data file for the mtline instance and then edit the model file mtline instance to point to the data file.

1. Start LMG.

lmg

The Transmission Line Model Generator window appears.

2. Specify the parameters that characterize the mtline instance.

For guidance, see "Modeling Transmission Lines Using the LMG GUI" in the *Spectre RF User Guide*.

3. In the Transmission Line Model Generator window, choose *File - LRCG file name*.

The LRCG File Name form appears.

4. In the *LRCG Name* field, specify a data file name. The program creates this file.

5. Click *OK*.

6. In the Transmission Line Model Generator window, click Calculate Parameters.

The calculated parameters appear in the top of the window.

7. (Optional) Choose *Options - Output file Control - RLCG file*.

This turns off the generation of a macromodel that the AMS Designer simulator does not use.

8. Click *Create Macromodel*.

The Macromodel Creation notice appears with a message similar to the following:

```
Per-unit-length RLCG file is generated and written to the file myw_line.dat
```

9. Click *OK*.

10. (Optional) In the Transmission Line Model Generator window, choose *File - Quit*.

The window closes.

11. Edit the model file (or files) that instantiates the [mtline](#) device so that you specify the `file` parameter of the instance with the absolute path to the data file (from [step 4](#)).

For example, the instance might look like this after editing:

```
mline (P_in P_out N_in N_out 0 0) mtline len=0.0762 romdatfile=""  
+ c=[1.600800e-10 -2.373800e-11 1.600800e-10]  
+ file="/hm/tiegsc/work/mtline/line.data"
```

At this point, the model file is ready for you to use.

Related Topics

- [Using mtline in a Schematic with AMS Simulator](#)

Design Configuration Using Mixed Languages

Both Spectre AMS Designer Simulator and Xcelium simulator with mixed-signal option allow you to simulate designs with connections such as SystemVerilog logic ports to electrical and SystemVerilog logic vector to a SPICE electrical output bus.

Related Topics

- [Features Enabled Using the -adv_ms Option](#)
- [SystemVerilog and AMS Extensions](#)

Verilog-AMS Connect Modules for VHDL-SPICE Connection

You can use the AMS Designer built-in CMs (`*_0_CE`) present in the AMSD built-in `connectLib` library instead of VHDL-AMS model in `std_logic` and electrical signal connections for VHDL-SPICE connections. This provides significant performance improvements for VHDL-SPICE applications.

You can find the `*_0_CE` connect modules in the Cadence hierarchy at:

```
<your_install_directory>/tools.<plat>/affirma_ams/etc/connect_lib/connectLib
```

To use the Verilog-AMS CMs for VHDL-SPICE connections, you need to compile the AMSD built-in connect element (CE) into a test case library, as shown below.

```
xrun -compile `cds_root` xrun`/tools/affirma_ams/etc/vhdlams_connectlib_samples/*.vhms
```

This feature does not change the use model of the `ce` statement in the `amsd` block and the CE report for VHDL-SPICE connections.

This feature is enabled by using the `xmelab` or `xrun` option `-use_cm`.

Currently, this feature supports only `std_logic` and electrical connections.

The following are not supported:

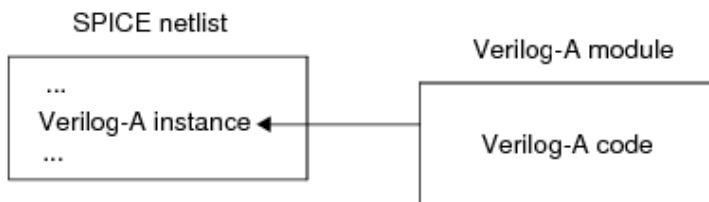
- VHDL real and SPICE electrical connection
- VHDL subtypes of `std_logic` and VHDL record types

Related Topics

- [Connecting VHDL Blocks to SPICE Blocks](#)
- [VHDL-SPICE Conversion Element Optimization](#)

Verilog-A Modules in SPICE Blocks

If you are using the AMS Designer simulator with the Spectre solver and the simulation front end (SFE) parser, you can use a `.ahdl_include` statement in a SPICE netlist to include behavioral or structural Verilog-A modules.



The `.ahdl_include` statement has the following format:

```
.ahdl_include " filename "
```

For `filename`, you can specify either a full or a relative path that resolves across your network to the file containing the Verilog-A modules. The file name must have a `.va` extension to indicate that the included modules are Verilog-A language models.

When you use the `.ahdl_include` statement to include a Verilog-A module in a SPICE netlist, the name of the instance that instantiates the Verilog-A behavioral module must not begin with `Y`.

Verilog-AMS Vector Buses with SPICE Subcircuits

You can connect Verilog-AMS vector buses directly to SPICE buses using the `sourcefile` property to specify the name of the SPICE file and the `sourcefile_opts` property to specify the Verilog-AMS to SPICE bindings. For details, see "sourcefile Property", "sourcefile_opts Property", and "The Port Mapping File".

Related Topics

- [Mixed-Signal Design Elaboration](#)
- [Analog Primitives and Subcircuits](#)

SPICE-on-Top

The AMS Designer simulator supports SPICE-on-Top configurations, where a SPICE file contains the top-level SPICE blocks or the top-level Verilog/SystemVerilog blocks.

To run a SPICE-on-Top design, you use the `-spicetop` option with the `xrun` command.

For example:

```
xrun xtop.scs foo.v <other xrun options> -spicetop
```

where:

- `foo.v` is verilog file in which a Verilog block is defined as top level module reference in spice-on-top.
- `xtop.scs` contains the SPICE blocks and an `amsd` block as follows:

```
amsd{  
    portmap module=foo  
    config cell=foo use=hdl  
    ie vsup=1.8  
}
```

The following example illustrates a SystemVerilog block in the Spice-on-Top configurations:

```
xrun sv_child.sv top.scs -spicetop -timescale 1ns/1ps
```

where, `top.scs` contains the SPICE blocks and an `amsd` block as follows:

```
//top.scs  
  
include "../top.sp"
```

```
amsd {  
    portmap module= sv_child  
    config cell=sv_child use=hdl  
    ie vsup=1.8  
}  
  
simulator lang=spectre  
tran tran stop=500ns  
  
//top.sp  
  
simulator lang=spice  
x1 a1 b1 c1 top_spice  
  
.subckt top_spice a b c  
v1 ( a 0) vsource dc=5  
v2 ( b 0) vsource dc=5  
  
x2 a b c sv_child  
.ends  
  
x3 a2 b2 c2 sv_child  
  
.subckt sv_child a b c  
.ends
```

And, the SystemVerilog block is defined in the `sv_child.sv` file, as follows:

```
module sv_child (in1,in2,out);  
input real in1,in2;  
output real out;  
  
always @(in1 or in2)  
begin  
out=in1 + in2;  
end  
  
initial $monitor ("Values in child::%g,\t%g,\t%g", in1, in2,out);  
endmodule
```

Related Topics

- [SPICE-in-the-Middle](#)
- [amsd Block](#)
- [portmap](#)

SPICE-in-the-Middle

The AMS Designer simulator supports the use of SPICE blocks in the middle of two Verilog blocks or two VHDL blocks.

A SPICE-in-the-middle arrangement consists of a hierarchy in which a Verilog-AMS block instantiates a SPICE block that, in turn, instantiates a Verilog-AMS block.

For example:

```
// top.v -- Verilog top
module top;
wire [0:1]a, b, out;
reg a1, b1;
ana_gate ana_gate (a, b, out) ;
endmodule

// ana_gate.sp -- SPICE in the middle
.subckt ana_gate inleft[0] inleft[1] inright[0] inright[1] out[0] out[1]
xnand1 inleft[0] inleft[1] out1 out2 out1 nand2
xnand2 inright[0] inright[1] out1 out2 out2 nand2
.ends ana_gate

// nand2.v -- Verilog leaf
module nand2( a, out );
input [0:3] a;
output out;
assign out = ~(a[0] & a[1]);
endmodule
```

If you are using the AMS Designer simulator with the Spectre solver and the simulation front end (SFE) parser, you can specify the Verilog-AMS module that you instantiate in the SPICE block.

To do this:

- Specify the `use` assignment in a `config` statement in an `amsd` block.

For example, assuming the `nand2.v` reference file is in the current working directory:

```
include "../ana_gate.sp"

amsd{

    portmap subckt=ana_gate autobus=yes busdelim="<>"
    config cell=ana_gate use=spice
    portmap module=nand2 reffile=nand2.v busdelim="<>"
    config cell=nand2 use=hdl
}
```

If you have a SPICE-in-the-middle arrangement using Verilog-A, you include the Verilog-A file using a Spectre `ahdl_include` statement, and you do not specify `use=hdl` using a `config` statement. The software uses analog elaboration to process Verilog-A design units.

If you use the `reffile` parameter on a `portmap` statement to specify custom port bindings for a SPICE-in-the-middle arrangement, the reference file must satisfy the following requirements:

- The reference file must compile standalone.
- If the reference file contains any include files, directives, or macros, they must be available and compile successfully.
- If the reference file contains any external language constructs, they must compile successfully without any special command-line options (such as `-sv`).

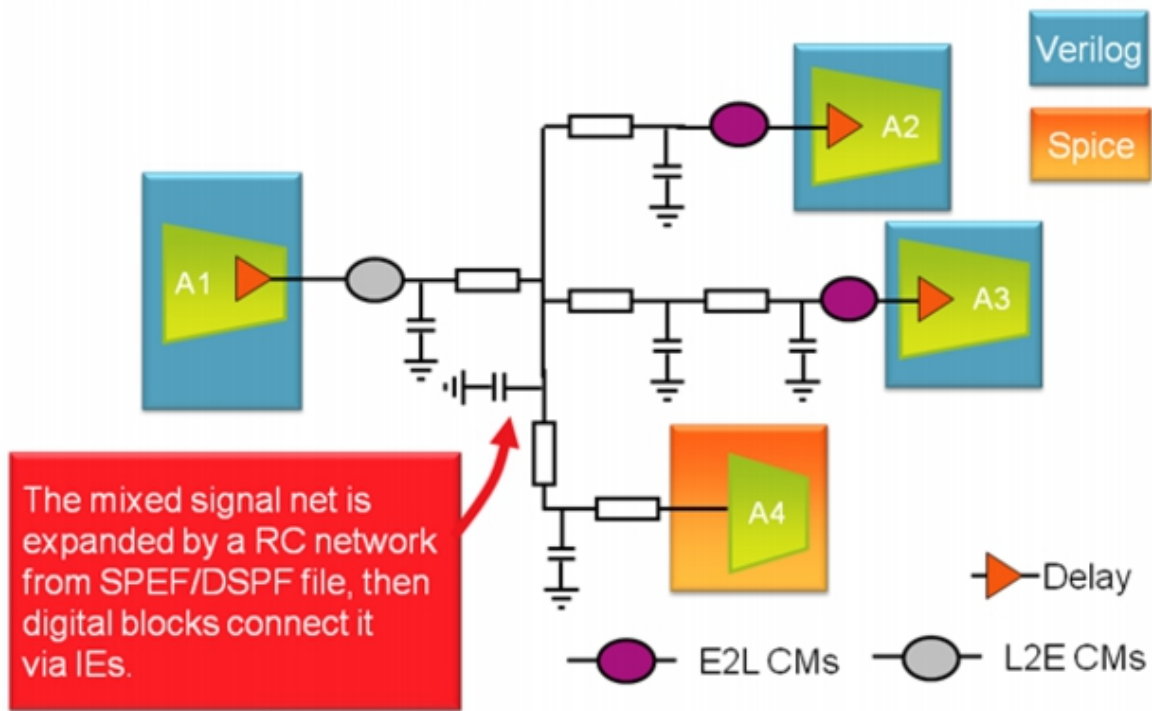
Related Topics

- [config](#)
- [portmap](#)
- [reffile](#)
- [SFE](#)
- [use](#)
- [Specifying DSPF and SPEF Stitching on Analog and Mixed-Signal Nets](#)
- [SPICE Blocks in the Middle of VHDL Blocks](#)

Specifying DSPF and SPEF Stitching on Analog and Mixed-Signal Nets

The parasitic RC network may be stitching onto an analog and mixed-signal net as it works in a pure SPICE netlist. When you change some blocks from Spice to Verilog in a post-layout design that is based on DSPF/SPEF stitching flow, the stitching engine in the analog solver tries to connect the instance pin of the RC network to the Verilog port. These blocks are redirected to connect the analog port of the connect modules (CM s), as shown in the figure.

Figure 7-1 RCs from SPEF are Expanded on Mixed-Signal Nets



To correctly specify SPEF or DSPF stitching on a mixed-signal net in AMS-APS, perform the following steps:

1. Set up the SPEF or DSPF file in the SPICE netlist, as follows:
`.usim_opt spef= "<instance|subckt> file"`
2. Set up the SDF file in the `initial` block or compile it and use the `-SDF_Cmd_file` option.
3. Set `mode=split` in the `ie` card, to keep the expanded RC net connection correct. For example:

```
amsd{
ie vsup=1.8 txdel=100n vthi=0.9 rlo=1 rhi=1 mode=split
portmap subckt=test_sim autobus=yes busdelim="[]" interconnect=electrical
config cell=test_sim use=spice
}
```

If you need to stitch the SPEF to the Verilog-to-Verilog connection in the SPICE block, you must set `interconnect=electrical`. This is because xmelab optimizes this node to logic discipline even if it lies in the SPICE block.

4. Set up the `-amsspef` option in the `xrun` arguments to control this feature, as shown below.

```
xrun ./*.v \
./config.scs \
./cds_globals.vams \
```

```
./top.vams \  
-input ./probe.tcl \  
-sdf_cmd_file ./command_sdf \  
-amsspef
```

This feature is supported in AMS-APS and AMS-Spectre solvers.

The behavior of AMS-APS stitching is similar to standalone APS stitching. For example, the `*.spfrpt` file is dumped into the simulation directory.

Related Topics

- [SPICE-in-the-Middle](#)
- [SPICE Blocks in the Middle of VHDL Blocks](#)

SPICE Blocks in the Middle of VHDL Blocks

The AMS Designer simulator supports most VHDL block types in a design, including multi-field records, scalars, and arrays. The supported port types include VHDL pure digital entities with standard logic, ports, or port vectors; VHDL user type ports; VHDL real signal ports; VHDL-AMS entities with analog terminal; and digital ports.

A SPICE-in-the-middle arrangement within VHDL blocks consists of a hierarchy in which one or more SPICE blocks are placed between two VHDL blocks. In this arrangement, one VHDL block instantiates a SPICE block that, in turn, instantiates another VHDL block.

The `reffile` parameter is not required for a VHDL block that is instantiated in a SPICE block.

In the following example, a SPICE block (`dummy_spice`) is placed between two VHDL blocks (`top.vhd` and `leaf.vhd`):


```
-- top.vhd -- VHDL top
entity top is
end entity top;

architecture a_top of top is
    signal v1, v3, v5 : real;
begin
    test : entity work.dummy_spice port map (v1, v3, v5) ;
end
architecture a_top;

// subckts.m -- SPICE in the middle
.subckt dummy_spice v1 v2 v3
x1 v1 leaf p=1
x2 v2 leaf p=3
x3 v3 leaf p=5
.ends dummy_spice

-- leaf.vhd -- VHDL leaf
entity leaf is
    generic (p : real := 2.0);
    port (signal n : out real);
end entity leaf;

architecture a_leaf of leaf is
begin
    n <= p;
end
architecture a_leaf;
```

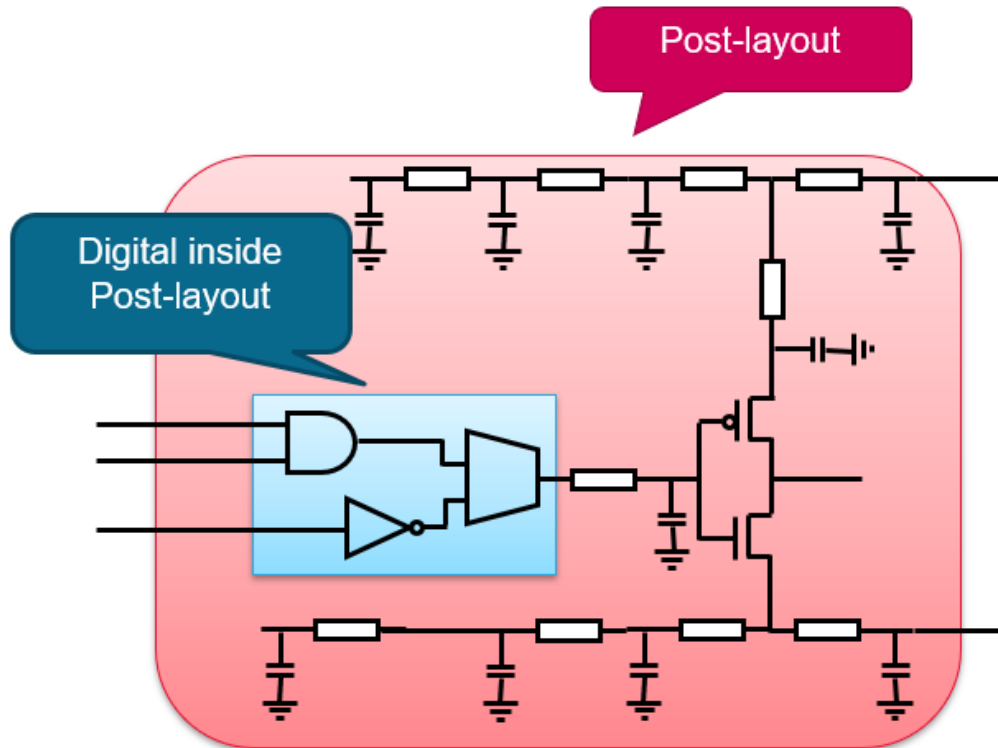
Related Topics

- [SPICE-in-the-Middle](#)
- [Specifying DSPF and SPEF Stitching on Analog and Mixed-Signal Nets](#)

Using DSPF-in-the-Middle

AMS Designer allows usage of a digital module inside DSPF for post-layout simulation. Using DSPF-in-the-middle speeds up mixed-signal post-layout simulation and it enables running verification as early as possible.

The following diagram illustrates a DSPF subcircuit.



In this diagram, the outer block contains a circuit topology, `myRingOsc16`. The inner blue block represents a sub-subcircuit, `myBufL8`. Consider that you need to replace `myBufL8` with a Verilog module because it is faster to compute Verilog modules. To bind `myBufL8` as Verilog module instead of a DSPF subcircuit, perform the following steps:

1. Open the `.scs` file.
2. Add the `dspf_include` command with name of the DSPF file and the `blackbox` keyword. Add other keywords, as required.

```
dspf_include "../dspf/myRingOsc16_BB.dspf" blackbox="myBufL8" bus_delim="<> []"
```
3. To change the binding from SPICE to Verilog, point to the DSPF in the `config` statement and switch the binding inside DSPF.

```
//AMSD Block
```

```
amsd{
  //IE Definition Default
  ie vsup-1.8 discipline=logic
  //Partitioning
  portmap module=myBufL8 reffile=muBufL8.v porttype=order
  config inst=myRingOsc16.TB.I0.XI0\XI0\XI0 use=hdl
}
```

Switching of binding is supported only when the blackbox feature is enabled and DSPF is hierarchical.

Similar to the SPICE-in-the-Middle flow, you can further configure the blackbox instance `myRingOsc16.TB.I0.XI0\XI0\XI0` with mixed-signal modules, within an `amsd` block

Related Topics

- [amsd Block](#)
- [SPICE-in-the-Middle](#)

-keepcase4use5x

While VHDL-AMS is case-insensitive, the DFII 5X library database is case-sensitive. For new cells and new models, use primarily lower case characters for the `cell name`, `entity names`, `generic names`, `port names`, and `architecture names`.

Use the `ncvhdl -keepcase4use5x` option to compel the `ncvhdl` compiler to retain case sensitivity during compilation. The `-use5xkeepcase` option keeps the case as for a VHDL cell name in a 5X library.

For example, see the following `Mixed_case` cell in VHDL code.

```
entity Mixed_case
...
end entity Mixed_case;
```

This code creates a `Mixed_case` cell instead of a `mixed_case` cell in a 5X library database.

In addition, you can also use the UNIX environment variable `CDS_ALT_NMP`. When you set it to `MATCH` as shown below:

```
setenv CDS_ALT_NMP MATCH
```

AMS-D in HED and AMS-D in ADE compile using the following command:

```
ncvhdl -specificunit Abc
```

instead of

```
ncvhdl -specificunit \Abc\
```

If you have a case-sensitive cell name matching the entity name, you should set the UNIX variable `CDS_ALT_NMP_MATCH` variable and use the `ncvhdl -keepcase4use5x` option.

Related Topics

- [Connections from VHDL and VHDL-AMS Blocks to Verilog and Verilog-AMS Blocks](#)
- [Mapping Verilog-AMS Disciplines to VHDL-AMS Natures](#)

Usage of Port Expressions when Connecting to Analog Blocks

You can use port expressions when connecting Verilog to SPICE or an analog block in your design when using the AMS Designer simulator. Port expressions can contain any of the following:

- Register or constant expression

```
electrical gnd;  
reg reg1 = 1'b1; // register driver connecting a SPICE primitive  
my_res r1(reg1, gnd); // SPICE subcircuit specified in prog.cfg file
```

- Digital concatenation

```
reg [0:2] reg_3 = 3'b101;  
reg [0:1] reg_2 = 2'b01;  
reg reg_1 = 1'b1;  
logic logic_1 ;  
logic [0:2] bus_3;  
logic [0:1] bus_2;
```

```
assign logic_1 = reg_1;

analog_child child1({3'b101, reg_1});
analog_child child2({1'b1, reg_3});
analog_child child1({2'b01, bus_2});
analog_child child2({bus_3, 1'b1});
analog_child child1({reg_3[0:1], bus_3[2], reg_3[2]});
analog_child child2({reg_3[1:2], bus_2[0:1]});
```

- **Operator expression**

```
reg r1 = 1'b0;
reg r2 = 1'b1;

analog_child a1 ( r1|r2 );
```

- **Analog nets and digital expressions inside a digital concatenation**

```
electrical ana_1;
wire w1;
analog_child child1({1'b1, ana_1});
analog_child child2({1'b1, w1});
```

- **Multiple or complex concatenation**

```
logic [0:1] a;
reg [0:1] b;
electrical ana;

{2{2'b10, a, b, ana}}
{{2'b10, a, b, ana}, {2'b10, a, b, ana}}
{{2{2'b10, a, b, ana}}, a, ana, {b, a, 2'b01}}
{2{{2'b10, a, b, ana}, ana, {2'b10, a, b}}}
```

- **Parameter expressions**

```
parameter real c=1.0;
```

```
spice_block i1(c, top.I1.c);
```

- Out-of-module reference

```
parameter p=2`b10;  
reg [2:3] r;
```

```
spice_block s1(top.d1.r, top.d1.l, top.d1.p);
```

Port expressions must contain neither any multiply-recursive concatenation constructs nor any concatenations with real parameter parts (such as `parameter in_real = 5.0; analog_child child1( {2'b01, in_real} );`).

When you have a digital expression (constant expression, register, or operator expression) that connects to an analog port, the elaborator automatically changes the port direction to `input` and inserts a logic-to-electrical (`l2e`) interface element.

When you have digital behavior in a port expression that connects to an analog port, the elaborator forces the operand to be logic (digital domain). For example:

```
wire w1,w2; // Elaborator forces w1 and w2 to be digital nets analog_child a1 ( w1|w2 );
```

Related Topics

- [Guidelines for Binding Ports by Name or Order](#)
- [Port Bindings between Verilog and SPICE](#)

SPICE Nets inside a Verilog Design

You can use the following special instances to access SPICE nets in a Verilog design:

Instance Type	Description
<code>cds_spice_a2d</code>	Analog-to-digital connection from SPICE to Verilog (analog drives)
<code>cds_spice_d2a</code>	Digital-to-analog connection from Verilog to SPICE (digital drives)

cds_spice_bidir	Bidirectional connection between Verilog and SPICE
cds_spice_a2a	Analog-to-analog connection from Verilog to SPICE (such as driving a SPICE port from the top level)

To use these instances, specify the full hierarchical path to the SPICE net you want to access as the parameter of the instance and the Verilog net as the port connection. Here are some examples:

```
cds_spice_d2a #("test.top.I0.I1.X1.X2.drive") d2a1(drive);
cds_spice_a2d #("test.top.I0.I1.X1.X2.read") a2d1(read);
cds_spice_d2a #("TestBench.SPICEblock.X2.regf") D2A1 (fault);
cds_spice_a2d #("TestBench.SPICEblock.X2.rega") A2D1 (chk1);
cds_spice_a2d #("TestBench.SPICEblock.X2.regb") A2D2 (chk2);
cds_spice_a2d #("TestBench.SPICEblock.X2.regc") A2D3 (chk3);
```

Consider the following digital testbench code (Verilog):

```
module TB;    // the testbenchdut

DUT(in1, in2); // instantiation of the device under test (the DUT)

reg in1, in2;    // drivers of the DUT
wire chk1, chk2, chk3; // signals used to check inside the DUT for
                      // erroneous conditions

reg fault;    // a signal used to generate a fault

initial begin // assign values to the DUT drivers
    in1 = 1'b1; in2 = 1'b1;
end

initial begin // generate a fault condition at 20 ticks
    TB.DUT.fault = 1'b0;
    #20 fault = 1'b1;
end

// monitoring of checks - an error is printed if they trigger
always @(chk1) $strobe("error chk1 triggered!\n");
always @(chk2) $strobe("error chk2 triggered!\n");
always @(chk3) $strobe("error chk3 triggered!\n");

// These statements make connections from the local testbench
// signals to the DUT. It might be convenient for the
// testbench to alias signals in the DUT to local
// signals in the testbench so that if the pathnames get
// changed in the design process, you can adapt the testbench
// easily by just changing this block and
// not the complex logic of the testbench
.assign TB.A.DD.f = fault;
assign chk1 = TB.A.DD.a;
assign chk2 = TB.A.DD.b;
assign chk3 = TB.A.DD.c;

endmodule
```

Perhaps the Verilog code for the **DUT** looks something like this:

<pre> module dut(in1, in2); blockA A(in1); blockB B(in2); endmodule module blockA(in); blockC CC(in); blockD DD(in); endmodule module blockB(in); blockE EE(in); endmodule module blockCC(in); endmodule module blockDD(in); reg a,b,c,d; reg f; ...etc endmodule module blockEE(in); endmodule </pre>	<pre> // Design Structure // // TB (the testbench) // // DUT // // ----- // // A B // // ----- EE // // CC DD </pre>

Perhaps the SPICE representation of `blockA` looks something like this:

<pre> subckt blockA(in) X1 in blockCC X2 in blockDD ends subckt blockDD(in) X1 in a X2 a b blockXXX X3 b c blockYYY X4 c d f blockZZZ ends </pre>	<pre> // Design Structure with blockA as SPICE // // TB (the testbench) // // DUT // // ----- // // A B // // ----- // EE // X1 X2 </pre>

To account for a SPICE block substitution in the testbench code, you could have the following:

...

```
// No changes to the testbench code before this point...
`ifdef AMS_MODE                // Add these lines for SPICE block substitution

cds_spice_d2a #("TB.A.X2.f") D2A1 (fault);
cds_spice_a2d #("TB.A.X2.a") A2D1 (chk1);
cds_spice_a2d #("TB.A.X2.b") A2D2 (chk2);
cds_spice_a2d #("TB.A.X2.c") A2D3 (chk3);

`else                          // pure-digital configuration

assign TB.A.DD.f = fault;
assign chk1 = TB.A.DD.a;
assign chk2 = TB.A.DD.b;
assign chk3 = TB.A.DD.c;
`endif
...
```

The following table outlines the effective module definitions for these instance types as well as the resulting interface elements (IEs) the software inserts in your design:

Instance Type	Effective Module Definition	Interface Element
cds_spice_a2d	<pre>module cds_spice_a2d(e); output e; electrical e; parameter spicenet = "null"; endmodule</pre>	analog-to-digital
cds_spice_d2a	<pre>module cds_spice_d2a(e); input e; electrical e; parameter spicenet = "null"; endmodule</pre>	digital-to-analog
cds_spice_bidir	<pre>module cds_spice_bidir(e); inout e; electrical e; parameter spicenet = "null"; endmodule</pre>	bidirectional

cds_spice_a2a	<pre>module cds_spice_a2a(e); input e; electrical e; parameter spicenet = "null"; endmodule</pre>	none

Related Topic

[Probing SPICE Ports](#)

Reusing Mixed-Language Testbenches

When reusing mixed-language testbenches, you can specify how you want the software to manage out-of-module references in digital statements when you substitute a SPICE block for a purely digital (Verilog) block. You can either use [compiler directives](#) around portions of your Verilog source code, or use a [command-line option](#) to specify your preference globally (without having to edit your Verilog source code). See the following topics for details:

- [Using Command-Line Options to Manage Out-of-Module References to SPICE](#)
- [Using Compiler Directives to Manage Out-of-Module References to SPICE](#)
- [Using Bus Delimiters with Out-of-Module References to SPICE](#)

Using Command-Line Options to Manage Out-of-Module References to SPICE

You can use the `-autospiceoomr` and `-autospiceignore` command-line options to specify how you want the software to manage out-of-module references (OOMRs) in digital statements when you substitute a SPICE block with a purely digital (Verilog) block.

When you use the `-autospiceoomr` option, the software automatically converts the SPICE OOMRs to appropriate digital values and vice versa. And, the software automatically prefixes X to spice instance names in OOMRs to spice ports or internal nets.

The software does a mapping of a digital OOMR having Verilog hierarchy instance names to an equivalent SPICE ports/nets. The software adds the prefix 'X' to spice instance names, if the instance names are specified without the prefix "X" in the verilog testbench.

However, if the spice instance name begins with "x" and also has the prefix "X", the software picks the instance name only. For instance, `Xxa2` in the example: `top.a1.Xxa2.a3.d: top.a1.xa2.Xa3.d`.

For mixed circuit netlist, software adds prefix "X" to spectre instance name only if the instance name begins with "x" and the equivalent instance name is specified without the "x" in the Verilog testbench.

Consider the following example of a Verilog testbench where SPICE subckt, `anal` and Spectre subckt, `mycell` are instantiated:

```
module top();
    anal a1 (out1);
    mycell m1 (a, b);
    dig_monitor D1 ();
endmodule

module dig_monitor();
    initial begin
        // we want to plot the d bus values after E2L conversion
        #10 $display( "=====%M internal bus0= %h", top.a1.Xxa2.a3.d,,
top.m1.i1.i22.pa,,top.m1.i1.i22.pb);
    end
endmodule
```

The following example illustrates the spice netlist and spectre netlist where the sub-circuits `anal` and `mycell` are defined:

anal.sp:

```
.subckt ana3 out1
I0 0 d 1.5u
R0 d 0 1meg
.ends
.subckt ana1 out1
Xxa2 out1 ana2
.ends
.subckt ana2 out1
a3 out1 ana3
.ends
.tran 1n 100u
.probe V(*)
.end
```

mycell.scs:

```
subckt mycell pa pb
```

```
i1 (pa pb) mycell2
ends

subckt mycell2 pa pb
i22 (pa pb) mycell3
ends

subckt mycell3 pa pb
i31 (pa 0) vsource dc=2
i32 (pb\[0\] 0) vsource dc=2
i33 (pb\[1\] 0) vsource dc=2
ends
```

And, in the AMSD control file, `a1` and `m1` are replaced by sub-circuits, `ana1` and `mycell`, respectively as follows:

```
include "mycell.scs"

amsd{
    ie vsup=1.8
    portmap subckt=ana1
    config cell=ana1 use=spice
    portmap subckt=mycell
    config cell=mycell use=spice
}
```

Note that in the above Verilog testbench the spice instance `xa2` is specified with a prefix "X" and the `xa3` is specified without the prefix "X". When you use the `-autospiceoomr` command-line option, the software automatically does the following mapping for the spice instance names:

```
top.a1.Xxa2.a3.d: top.a1.xa2.Xa3.d:
top.m1.i1. i22.pa: top.m1.i1. i22.pa:
top.m1. i1. i22.pb: top.m1.i1. i22.pb:
```

When you use the `-autospiceignore` command-line option, the software ignores any digital statements that contain OOMRs to SPICE blocks. You can specify one of the following logic values to the `-autospiceignore` option: `x|z|0|1`.

In the following example, if you specify `-autospiceignore x`, assertion that is triggered at time 0 as `d1.x` is forced to `x`. And, if you specify the value as `-autospiceignore 1`, assertion that is not triggered as `'1.x` is forced to `1`.

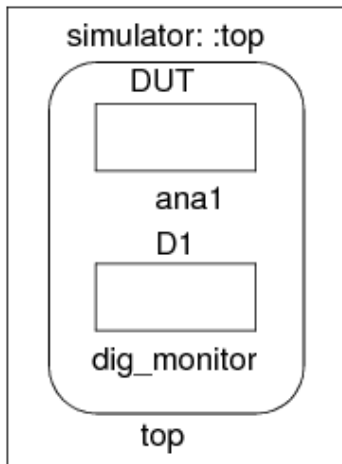
```
module tb;
wire w;

...
//SPICE block
dut d1(w);
...
```

```
assign w = 1'b1;
always @(w) begin
assert (d1.x != 1'b0) else begin
t = $time;
$error("assert failed at time %0t", t);
end
end
...
endmodule
module dut(x);
input x;
...
endmodule
```

If both `-autospiceoomr` and `-autospiceignore` options are specified, `-autospiceignore` option gets the higher priority.

Consider the following example:



Digital Verilog Code for top and dig_monitor

```
`timescale 1ns/1ns
module top();
    ana1 DUT ();
    dig_monitor D1 ();
endmodule

module dig_monitor();
    wire top;
    initial begin
        #5;
        $display( "=== %M ");
        $display( "=== internal %d, top.DUT.clk;
    end

    always @(top.DUT.clk) begin
        $display( ">> Instance %M top.DUT.clk event detected
                  %d at t=%f ns", top.DUT.clk, $time);
    end
endmodule
```

DUT is replaced by a SPICE subcircuit in the AMS control file, as follows:

```
//amscf1.scs file
include "source/spice/ana1.sp"
amsd{
    ie vsup=1.8
    portmap subckt ana1
```

```
config cell=anal use=spice
}
```

When the simulation is run using the `xrun` command

```
xrun -clean -mess source/verilog/top1.v -timescale 1ns/100ps -iereport amscf1.scs
```

the simulator generates an error related to OOMT `top.dut.clk`.

When you use the `-autospiceignore` command-line option with `xrun`, the simulator ignores the SPICE OOMR references and the simulation runs successfully.

When you use the `-autospiceoomr` command-line option with `xrun`, the simulator automatically converts the SPICE OOMR to a digital value. For example, the IE report will display the following:

Automatically inserted instance: `cds_autospiceoomr.top_DUT_clk__E2L_2__electrical`
(merged):

```
connectmodule name: E2L_2,
inserted across signal: top_DUT_clk
and ports of discipline: electrical
Sensitivity infomation:
    No Sensitivity info
Discipline of Port (Ain): electrical, Analog port
Discipline of Port (Dout): logic, Digital port
Drivers of port Dout: No drivers
Loads of port Dout: No loads
```

IE Report Summary:

```
E2L_2 ( electrical input; logic inout; ) total: 1
```

```
-----
Total Number of Connect Modules                                total: 1
```

You can also use the `-ignore_spice_oomr` and `-default_spice_oomr` command-line options to manage out-of-module references in digital statements when you substitute a SPICE block for a purely digital (Verilog) block. These options affect the following digital statements:

- `if` statements
- `$display` or `$monitor` statements
- Procedural assignments, including blocking and nonblocking
- `force` and `release` procedural statements
- Continuous assignments
- Sequential blocks (where the out-of-module reference to SPICE is in a delay or event control expression)

The `-ignore_spice_oomr` and `-default_spice_oomr` command-line options will be removed in a future release.

See also ["Using Compiler Directives to Manage Out-of-Module References to SPICE."](#)

Using Compiler Directives to Manage Out-of-Module References to SPICE

You can use the ``ams_testbench_reuse_ignore` and ``ams_testbench_reuse_default_value` directives to specify how you want the software to manage out-of-module references in digital statements when you substitute a SPICE block for a purely digital (Verilog) block. You can use these directives in conjunction with the following statements:

- `if` statements
- `$display` or `$monitor` statements
- Procedural assignments, including blocking and nonblocking
- `force` and `release` procedural statements
- Continuous assignments
- Sequential blocks (where the out-of-module reference to SPICE is in a delay or event control expression)

When you use the ``ams_testbench_reuse_ignore` directive, the software ignores any digital statements that contain out-of-module references to SPICE blocks. For example:

```
`ams_testbench_reuse_ignore
if ( spice_oomr_ref )
    $display("1");
else
    $display("2");
$display("3");
`end_ams_testbench_reuse_ignore
```

In this example (above), the software ignores `$display("1")` and `$display("2")` , but `$display("3")` remains active.

When you use the ``ams_testbench_reuse_default_value` directive, the software assigns a default value (1'b_x) if it encounters a digital statement that contains an out-of-module reference to a SPICE block.

For example, if you substitute a SPICE block such that `testbench.p1.p0` becomes an out-of-module reference in the following testbench code, the ``ams_testbench_reuse_default_value` directive

causes the software to set the value in the statement to 1'b_x.

```
module testbench ();
    ...
    wire vcoclk, clock_2, clock_1, clock_0, net036, p0;
    ...
    `ams_testbench_reuse_default_value
    initial begin
        $monitor (testbench.p1.p0);
    end `end_ams_testbench_reuse_default_value
    ...
    pll_top p1(refclk, reset, vcoclk, clock_2, clock_1, clock_0, net036, p0,
               clk_p0_1x, clk_p0_4x);
endmodule
```

See also ["Using Command-Line Options to Manage Out-of-Module References to SPICE."](#)

Using Bus Delimiters with Out-of-Module References to SPICE

You can use the `-autospice_addbusdelim` option to specify additional bus delimiters apart from the default ones, "`[] <>`" to enable spice out-of-module references (OOMRs). It allows you to use the bus delimiters in all SPICE buses/ports including the ones in ports at HDL boundary as well as internal ones.

The `-autospice_addbusdelim` option works only for auto SPICE OOMRs; therefore, you must specify the the `-autospice_addbusdelim` option along with the option as follows:

```
xrun -autospiceoomr -autospice_addbusdelim "_ __ && empty ( % %%"
```

The single delimiter `_` is used between bus base-name and index; the pair delimiters, `__`, `&&`, and `%%` are used around the index; and, `empty` is used for NULL delimiter- for example "abc0, abc1".

The following example illustrates that `_` works for vector `i` in subckt, `spice_middle`; `__` works for `i0`; `&&` works for `i1`; `empty` works for `abc`; `(` works for `i2`; `%` works for `i3`; `%%` works for `i4`:

```
// amscf.scs

amsd{
    ie vsup=1.8
    portmap subckt=spice_middle file="./spice_middle.pb"
    config cell=spice_middle use=spice
}

// top.v
module top;
    wire [0:3] i;
```

```

    wire [0:3] o;
    spice_middle x0 ( i, o );
    initial #3 $display("==== ",
top.x0.i,,top.x0.out,,top.x0.i0,,top.x0.i1,,top.x0.i2,,top.x0.abc,,top.x0.i3,,top.x0.i4);
endmodule

// spice.scs

subckt spice_middle ( i_3 i_2 i_1 i_0 out\<3\> out\<2\> out\<1\> out\<0\>)
v19 ( i4\%3\% 0) vsource dc=5
v18 ( i4\%2\% 0) vsource dc=5

v17 ( i3\%3 0) vsource dc=5
v16 ( i3\%2 0) vsource dc=5

v15 ( abc1 0) vsource dc=5
v14 ( abc0 0) vsource dc=5

v13 ( i2\2 0) vsource dc=5
v12 ( i2\1 0) vsource dc=5

v11 ( i1\&1\& 0) vsource dc=5
v10 ( i1\&0\& 0) vsource dc=5

v9 ( i0_3_ 0) vsource dc=5
v8 ( i0_2_ 0) vsource dc=5

v7 ( i_3 0) vsource dc=5
v6 ( i_2 0) vsource dc=5
v5 ( i_1 0) vsource dc=5
v4 ( i_0 0) vsource dc=5

v3 (out\<3\> 0) vsource dc=5
v2 (out\<2\> 0) vsource dc=5
v1 (out\<1\> 0) vsource dc=5
v0 (out\<0\> 0) vsource dc=5
ends

```

Values Associated with an Analog Object

You can use the function `cds_get_analog_value` to fetch the voltage, current, input, output, or power values associated with an analog object. This function is especially useful while writing testbenches and assertions for a design.

The `cds_get_analog_value` function is meant to be used only on objects belonging to a continuous domain. It can be called from within Verilog, Verilog-A, SystemVerilog (SV), or Verilog-AMS scope,

and always returns a real number.

Following is the syntax of the `cds_get_analog_value` function.

```
real $cds_get_analog_value (hierarchical_name, [optional index, [optional  
quantity qualifier]])
```

Or

```
real $cgav (hierarchical_name, [optional index, [optional quantity qualifier]])
```

where,

- `cgav` is an alias of `cds_get_analog_value`. They have an identical signature.
- The object referred to by `hierarchical_name` must exist and must be owned by the analog solver. It must be a scalar object. `hierarchical_name` can be a relative or absolute path.

You can check whether the object referred to by `hierarchical_name` meets these conditions by using the helper functions `cds_analog_is_valid`, `cds_analog_exists`, and `cds_analog_get_width`.

The hierarchical name can refer to a Verilog-AMS context or Spectre/SPICE (a node or instance from a Spectre/SPICE format netlist) context.

For the Spectre/SPICE netlist formats, bus or vector object notation is currently not supported. In the analog context, bus can be expanded as bits. If `<>` are bus delimiters, `<` and `>` cannot be used in identifier names in hierarchical path in Verilog/SystemVerilog. Verilog and SystemVerilog allow these illegal characters to be used in a name by escaping the identifier.

In the hierarchical name, when a bus is referred to in the Spectre/SPICE context, escape the bus node bit name.

For example, to fetch the voltage `tb.DUT.n<2>`, you will need to escape the node

```
.$cgav("tb.DUT.\S<2>", "potential");
```

- The index can be variable, reg, or parameters so long as their value evaluates to an integral constant.

Index applies only for vector objects.

Example:

```
$cgav("tb.DUT.n", 2, "potential").
```

This statement probes voltage of the vector bit node. The return value is the voltage

```
V(top.DUT.n[2]).
```

Note that the `$cgav` index feature works only if the `top.n[2]` vector is in the Verilog-AMS context

or scope.

- The quantity qualifier can be `potential` , `flow` , `pwr` , or `param`. If none is specified, `potential` is assumed.

If any of the above conditions is not satisfied, the behavior of the `cds_get_analog_value` function will be undefined. To test that these conditions are satisfied by the analog object being referenced and to create more reusable testbench code with failsafe behavior, you can use the following helper functions:

- `cds_analog_exists` - Function to determine if the object is a member of the analog domain.

Signature:

```
int $cds_analog_exists(hierarchical_name, [optional index])
```

Return value:

1/0

(where 1 is true and 0 is false)

Arguments:

- `hierarchical_name` can be in pure Verilog syntax.
 - If `index` is not specified, the object is assumed to be `scalar`.
- `cds_analog_get_width` - Function to determine the width of an object

Signature:

```
int $cds_analog_get_width(hierarchical_name)
```

Return value:

width of the object

- `cds_analog_is_valid` - Function to test if the reference object can be probed using the `cds_get_analog_value` function.

Signature:

```
int $cds_analog_is_valid(hierarchical_name, [optional index, [optional quantity  
qualifier]])
```

Return value:

1/0

Arguments:

- `hierarchical_name` can be in pure Verilog syntax.
 - If `index` is not specified, the object is assumed to be `scalar`.
 - If quantity qualifier is not specified, it is assumed that the hierarchical name is a net.
- `cds_set_temperature` - Function to change the global circuit temperature parameter during simulation.

Signature:

```
$cds_set_temperature(expression)
```

Return value:

1/0

Argument:

`expression` should be a real-type expression that represents the temperature to be set in degree Celsius. You can use variables and parameters in the expression.

- `simprobe` - Function to allow probing of values within a sibling instance during simulation in analog context.

Signature:

```
$simprobe( <inst_name> (string), <param_name> (string), [ optional real valued  
expression ] )
```

Return value:

Double value of the parameter that is probed.

Arguments:

- `inst_name` the name of the sibling instance in the current parent instance.
 - `param_name` is the output variable to be probed in `inst_name`.
 - If a parameter is not found and the `optional real valued expression` is set, then that would be the output.
- `$simprobe$str` - Function to allow probing of values within a sibling instance during simulation in analog context.

Signature:

```
$simprobe$str( <inst_name> (string), <param_name> (string), [ optional string  
expression] )
```

Return value:

String value of the parameter that is probed.

Arguments:

- `inst_name` the name of the sibling instance in the current parent instance.
- `param_name` is the output variable to be probed in `inst_name`.
- If a parameter is not found and the `optional real valued string` is set, then that would be the output.

Strength-Based Interface Element (SIE)

Strength-based Interface Element (SIE) is a new simulation technology that accurately models and simulates the analog or digital interface in a mixed-signal design. In SIE, both strength and logic levels of a digital signal are converted with the impedance and voltage level of the Thevenin equivalent circuit representing the analog interface. As a result, the signal value and direction on the mixed-signal net are resolved automatically and dynamically.

SIE's simulation accuracy (including the timing and logic value on mixed-signal nets of the currently supported features) is fully backward-compatible with virtually the same simulation performance. SIE:

- supports all strength-based Verilog models on the analog or digital interface. For example:
 - **tran gate** - `tran` , `tranif1` , `rtranif0`
 - **wire** - `supply1` , `supply0`
 - **buffer**

```
assign (pull1, weak0) y = x;
buffif (pull1, strong0) B1 (y, x, c);
```
- maintains natural connection of all devices on the Verilog side of the connect module, instead of segregating drivers from receivers.
- eliminates the analog loading effect on Verilog `supply1` or `supply0` net.
- provides an easy design configuration-and-verification flow with various strength-based Verilog models.

The SIE mode is turned on by default. However, if you have a design that contains connect modules that do not support SIE, the simulation automatically runs in non-SIE mode and a warning message is displayed. The log file also indicates that the simulation is running in non-SIE mode. For designs that do not use `ie` statement or connect modules/connect rules, the log file does not display any information related to SIE.

To run the simulation in SIE mode, all connect modules inserted in the design must support SIE, otherwise, the simulator automatically enables the non-SIE mode. It is recommended that you configure your design with the `basic`, `full`, or `full_fast` connect rules (available in the installation) by using one of the following methods, as applicable to your flow:

- In ADE, click *Setup > Connect Rules*
- In the `xrun` command-line flow, use the `ie` statement in the `amsd` block. For example, `ie connrules=< conn_rule_name >.`
- In the `xmelab` or `xrun` command-line flow, specify the connect rules. For example, `ConnRules_18V_full_fast.`

The following new set of parameters and their default values for driver output resistance corresponding to the Verilog signal strength resistances are defined in the `full_fast` connect rules:

Parameter Name	Default Value	Corresponding Verilog Driver Signal Strength
<code>rsupply</code>	4	<code>supply</code>

rout	200	strong
rlo	rout	strong0
rhi	rout	strong1
rpull	1.5e3	pull
rlarge	9.0e3	large
rweak=	5.5e4	weak
rmedium	3.2e5	medium
rsmall	1.9e6	small
rz	1.0e7	HiZ

To customize the strength parameters for a design, it is recommended to specify such parameter values in the `ie` statement of the `amsd` block (see `ie`). To ensure that SIE works correctly, the strength resistance parameter values must be specified in ascending order from `rsupply` to `rz`, with resistances of consecutive strength levels being at least three-times apart, otherwise, the simulator will generate an error.

The connect modules that support SIE contain the `$SIE_input` system task that enables you to obtain the input value and strength of the digital drivers. The `$SIE_input` system task takes two arguments. The first argument can be any digital scalar net. The second argument depends on the nettype of the first argument.

- If the first argument is a scalar logic wire, the second argument should be `wire [7:0]`
- If the first argument is scalar wreal, the second argument should also be scalar wreal
- If the first argument is a scalar SV wreal, the second argument should also be a scalar SV wreal with the same nettype

For example:

```
$SIE_input(Din, Dval);
```

SIE is supported in all existing connect modules of `basic`, `full`, and `full_fast` connect rules available in the AMSD installation. SIE is also supported in the AMS-CPF and IEEE 1801 standard (UPF) flows.

To model the bidirectional behavior of WTRAN models with SV-wreal nettypes, you can use the `$cds_get_external_drivers` system task.

Related Topics

- [ie](#)
- [Writing Bidirectional Model Behavior](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)

PSL Assertions

Using the AMS Designer simulator, you can specify [PSL](#) assertions on `real` and Verilog-AMS `wreal` nets that connect to `electrical` nets. This methodology enables you to use PSL assertions with analog and mixed-signal blocks that you model using the Verilog-AMS `wreal` data type.

You must use the `-assert` option with the `xrun` or `xmvlog` command to enable PSL assertions.

Expressions involving `wreal` type objects that are explicitly declared can appear in PSL assertions in boolean expressions, clocking expressions, and as actual arguments in property and sequence instances.

```
wreal mywreal1, mywreal2;

reg clk;

// psl assert always ({mywreal1 > 4.4; mywreal2 < 6.6}) @(posedge clk);
```

Here is an example of specifying PSL assertions on a `wreal` net, `out1`, in a Verilog-AMS module:

```
`timescale 1ns/100ps

module INV ( out1, in1 );

    output out1;
    input in1;
    wreal in1, out1;

    real out1_reg;

    // psl out_eq_in_real_vams: assert
    //   never ( out1 < 1.25 && in1 < 1.25);

    // psl out_eq_in_real_vams2: assert
    //   never ( out1 < 1.25 && in1 > 1.25);

    always @(in1)
        if(in1 > 1.25)
            out1_reg = 0.0;
        else
```

```
    out1_reg = 2.5;

    assign out1 = out1_reg;

endmodule
```

For `electrical` nets, you assign the `electrical` net to a `real` or `wreal` variable, and then specify the PSL assertion on the `real` or `wreal` variable. You can also specify PSL assertions directly on `electrical` nets that are enclosed within access functions.

If PSL assertions are used on a mixed net, the assertion statement affects the discipline of the net. The simulator generates a warning for such scenarios. For example, the AMS simulator generates a warning message when a net is connected to a digital construct in the assertion statement. Discrete discipline is used for such nets during discipline resolution.

```
module top;
    ana_test ana_test(abc);
    //psl comparator_pos: assert always (((top.abc > 1.0) )) @(
posedge(top.ana_test.int_var)) ;
endmodule

module ana_test(ana_port);
    output ana_port ;
    electrical ana_port ;
    reg int_var =0;
    always #1000 int_var = ~int_var ;
endmodule

//psl comparator_pos: assert always (((top.abc > 1.0) )) @( posedge(top.ana_test.int_var))
;
```

In the above example, net `top.abc` connects to electrical port, but it will have a resolved discrete discipline because it is used in digital constructs in the assertion statement.

You can use the xmelab option `-amselabtrace assert` option to trace nets that are connected to digital constructs in an assertion statement.

Analog Event Functions for Assertion Clocking

Verilog-AMS analog event functions `cross` and `above` are supported as clocking events in PSL assertions. For example:

```
electrical sig1, sig2, sig3;
reg a, b;
// psl assert always ({V(sig1);a} | => {V(sig2);b}) @(cross(V(sig3)));
```

Support for PSL Built-In Functions

Analog expressions are allowed only within the `prev` PSL built-in sampled value function. It is an error to have analog expressions as arguments to built-in sampled value functions other than `prev`. For example:

```
// psl assert always ({V(sigout) > 0.5} | => {prev(V(sigout)) > 0.4}) @(cross(V(sigout)));  
// ok  
  
// psl assert always ({V(sigout) > 0.5} | => {stable(V(sigout)) > 0.4})  
@(cross(V(sigout))); // not allowed
```

Related Topic

- [Limitations of Using PSL Assertions](#)

Limitations of Using PSL Assertions

The following limitations exist when using PSL assertions with the AMS Designer simulator:

- Unclocked, partially clocked, or multiple-clocked PSL assertions involving analog expressions are not supported.

```
electrical sig1, sig2;  
reg a, clk, clk1, clk2;  
  
// unclocked - Not Supported  
// psl assert always (V(sig1) > V(sig2));  
  
// partially clocked - Not Supported  
// psl assert always {V(sig1); V(sig2)} | => {a} @(cross(V(clk)));  
  
// multiple clocked - Not Supported  
// psl assert always {V(sig1); V(sig2)} @(cross(V(clk1))) | => {a} @(cross(V(clk2)));
```

For such cases, use either an explicitly declared single clock or a default clock.

- Analog expressions that are not allowed to appear outside the analog block as per the Verilog-AMS LRM cannot appear in a PSL assertion statement. In the example below, the `assert` statement is not allowed because the analog operator `ddt` cannot appear outside an analog block.

```
// psl assert always ({ddt(V(sig1))} |-> {ddt(V(sig2))}) @(posedge clk);
```

- When port probes are used with the `<>` syntax, they cannot appear in a PSL assertion. For example, the following `assert` statement is not valid.

```
electrical in, out;
// psl assert always (I(<out>) > 0) && (I(<out>) < 2.3) @ (cross(I(<in>)));
```

- The assertion clock cannot be level-sensitive on an analog expression. For example, the following `assert` statement is not valid because the assertion clock is sensitive to the value change of `V(in)`.

```
// psl property P1(bitvector i) = always (S1(i) | => {i > 0.6}) @ (V(in)); // psl
assert P1
```

To overcome this limitation, you can use an analog event expression in the assertion clock, as shown in the following example.

```
// psl property P1(bitvector i) = always (S1(i) | => {i
> 0.6}) @ (cross(V(in)));
// psl assert P1
```

- If a PSL assertion involves a level-sensitive clock, none of the assertion signals are allowed to contain analog expressions. The following assertion is not valid because the assertion signal involves the voltage of node `sig2`, while the clock is level-sensitive.

```
electrical sig1, sig2;
reg a, b, clk;
// psl assert always ({V(sig1);a} | => {V(sig2);b}) @(clk);
```

- Nets that could potentially be coerced to `wreal` during elaboration because they are connected to a `wreal` net or an analog object cannot be used in PSL assertions.

```
module top;
wreal wr;
test mytest(wr);
endmodule

module test(w);
input w;
reg clk;
// not ok - 'w' will be coerced to wreal due to its connectivity to wr and cannot
be be used in a psl assertion
// psl assert always (w > 2.2) @(posedge clk);
endmodule
```

For such cases, explicitly declare the `wreal` nets that you want to include in a PSL assertion.

- Although PSL assertions appearing in verification units that are bound to modules can contain analog expressions and `wreal` objects, instance-bound verification units cannot contain analog expressions. For example:

```
module top;
    .....
    test mytest();
endmodule

module test;
    .....
endmodule

vunit myvunit(test) { // ok, verification unit involving analog expression is
                      // module bound
    // psl property P1 = ({V(sig1)} -> next (V(sig2)) @(cross(V(sig3))));
    //psl assert P1;
}

vunit myvunit(top.mytest) { // not ok, verification unit involving analog
                           // expression is instance bound
    // psl property P1 = ({V(sig1)} -> next (V(sig2)) @(cross(V(sig3))));
    //psl assert P1;
}
```

- PSL cannot be used within VHDL-AMS if the PSL assertion statement contains objects or expressions that are either owned or evaluated by the continuous time solver.

Related Topics

- [PSL](#)
- [PSL Assertions](#)

Reusing Testbenches and IP

AMS/ADE enables you to reuse an existing testbench/IP in other designs. This can be done in the following three ways:

- **Advanced Testbench Reuse:** With this mode, you can reuse a digital super top above the

AMSD HED configuration that you are simulating in ADE.

- **Export as IP Block:**

- **Using ADE GUI:** With AMS-Unified Netlist, you can export the binding information, which includes netlist, text files, and configuration, and then reuse this information later in other designs. To do so in ADE, select *Tools>Export>Export as IP Block*.
- **Using the `runams` command:** You can also export IP block using the in the `runams - export` command-line option.

- **Using Command-Line IP Selector (CLIPS):** In CLIPS, specify a testbench and the tool shows the design hierarchy. From the hierarchy, you can switch any instance to a HED configuration. Then, the tool calls AMS-UNL to export the HED configuration as an IP and assemble the IP together with the testbench.

CLIPS works with XCELIUM 17.10 s007 or higher versions.

This feature is available only with XCELIUM 17.04 or higher versions.

Related Topics

- [Reusing an Advanced Testbench](#)
- [Exporting IP Blocks in AMS UNL](#)

Connecting VHDL Blocks to SPICE Blocks

Direct instantiation of SPICE blocks within a VHDL scope is supported only using the `xrun` command flow. The three step method is not supported for VHDL-SPICE flows. Analogous to the Verilog-SPICE solution, AMSCB flow is required to specify SPICE blocks, boundary port maps and binding information. You can use the portmap and config mstatements in an amsd block to specify port bindings at VHDL-SPICE boundaries.

Here are some examples:

```
amsd{
  portmap subckt=dummy_spice autobus=yes refformat=vhdl
  config cell=dummy_spice use=spice
}
```

and:

```
amsd{
  portmap subckt=dummy_spice2 autobus=yes reffile=ref.vhd refformat=vhdl
```

```
config cell=dummy_spice2 use=spice
}
```

In the example immediately above, the `reffile` contains a VHDL module that defines the port bindings to use from a VHDL parent to a SPICE subcircuit or instance. The `config` statement specifies which SPICE cell (subcircuit) or instance.

You can have user-defined types, subtypes, and records where VHDL connects to SPICE.

Consider the following example of a 16x16-bit multiplier (`.SUBCKT mult16x16_spice`) with two 16-bit inputs (`A<15:0>` and `B<15:0>`), a clock input (`CLK`), and a 32-bit output (`P<31:0>`).

```

* SPICE file: "mult16.net"
***** This is a 16-bit x 16-bit parallel unsigned multiplier *****
*
* A<15:0> 16-bit multiplicand input
* B<15:0> 16-bit multiplier input
* P<31:0> 32-bit product output
* PRD<31:0> 32-bit product output befor the output register
* CLK clock input
* RegA 16-bit input register(postive edge triggered)
* RegB 16-bit input register(postive edge triggered)
* RegP 32-bit output register(postive edge triggered)
*
*
*          ----
*          16 |    |
* A<15:0> ----/-->|RegA|-----+
*                |    |         |
*          +----->|    |         v
*                |    |         ---- PRD<31:0>
*                |    |         |    |
*                |    |         | 16x16 | 32 |    |
*                |    |         |Multiplier|---/---|RegP|---/---> P<31:0>
*                |    |         | array   |    |    | 32
*                |    |         |    |         +--->|    |
*                |    |         ----
*                |    |         ^
*          16 |    |         |    |
* B<15:0> ----/-->|RegB|-----+
*                |    |         |
*          +----->|    |         |
*                |    |         |
*          CLK  --+-----+
*
*****

```

You can create a VHDL `reffile` such as the following that defines the directions for the interface elements between SPICE and VHDL:


```
-- VHDL reffile: "mult16x16_spice.vhd"

LIBRARY IEEE;
USE      IEEE.STD_LOGIC_1164.ALL;
ENTITY mult16x16_spice IS
    PORT (
        A : in std_logic_vector ( 15 DOWNTO 0 );
        B : in std_logic_vector ( 15 DOWNTO 0 );
        CLK : in  std_logic;
        P : out std_logic_vector( 31 DOWNTO 0 ));
END ENTITY mult16x16_spice;
```

Your VHDL testbench might look like this:

```
LIBRARY ieee;
USE ieee.math_real.ALL;
USE IEEE.STD_LOGIC_1164.ALL;
    USE      IEEE.ELECTRICAL_SYSTEMS.all;
LIBRARY STD;
USE STD.textio.all;

LIBRARY worklib;
USE worklib.ALL;

ENTITY top ISEND  top;

ARCHITECTURE bhv OF top IS
    SIGNAL sa : std_logic_vector (15 downto 0);
    SIGNAL sb : std_logic_vector (15 downto 0);
    SIGNAL sp : std_logic_vector (31 downto 0);
    SIGNAL sclk : std_logic;

BEGIN
    Iclk: ENTITY work.clk_gen
        PORT MAP ( clk => sclk);
        Iab: ENTITY work.a_b_gen
            PORT MAP ( clk => sclk, a => sa, b => sb);
        SPICE_DUT: ENTITY worklib.mult16x16_spice
            PORT MAP (sa , sb, sclk, sp);
        checka_d_da:  ENTITY worklib.DA_AD_GENERIC_CHECKS;
END ARCHITECTURE bhv;
```

In your control file, you include the SPICE file (which contains the `mult16x16_spice` subcircuit

definition) using an `include` statement, and you specify the port bindings you want the elaborator to use at VHDL-SPICE boundaries using `portmap` and `config` statements in an `amsd` block as follows:

```
include "mult16.net"

amsd{
    portmap subckt=mult16x16_spice autobus=yes refformat=vhdl
           reffile=source/mult16x16_spice.vhd
    config cell=mult16x16_spice use=spice
}
```

The `portmap` statement, above, indicates that you want the elaborator to use the port bindings you defined in the VHDL file, `mult16x16_spice.vhd`, and apply them to the SPICE subcircuit, `mult16x16_spice`. The `config` statement indicates that you want the elaborator to use the SPICE definition for the `mult16x16_spice` cell.

Further, you can specify conversion elements for the VHDL-to-SPICE connections using `ce` statements:

```
amsd{
    portmap subckt=mult16x16_spice autobus=yes refformat=vhdl
           reffile=source/mult16x16_spice.vhd
    config cell=mult16x16_spice use=spice
           ce name=worklib.std_logic2e dir=input type=std_logic genericmap="vsup 2.5"
           ce name=worklib.e2std_logic dir=out type=std_logic genericmap="vsup 2.5"
}
```

Related Topics

- [config](#)
- [portmap](#)
- [ce](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)

Connections from VHDL and VHDL-AMS Blocks to Verilog and Verilog-AMS Blocks

The AMS Designer simulator supports the following direct connections between VHDL or VHDL-AMS, Verilog or Verilog-AMS, and SIE-based Interface Elements:

- You can instantiate VHDL (digital) blocks in Verilog (digital) blocks and Verilog (digital) blocks in VHDL (digital) blocks.
- You can connect a Verilog `wire` to a VHDL `real` (when the VHDL `real` is below); the software treats the `wire` as a Verilog-AMS `wreal` signal.

When you instantiate a VHDL block containing `real` signal ports on a schematic, you create a connection of VHDL `real` signals to Verilog-AMS wires from above. In such a situation, the simulator coerces the Verilog-AMS `wire` to become a digital `wreal` to integrate such VHDL blocks into the AMS flow.

- You can connect a VHDL-AMS `real` signal to a Verilog-AMS `wire`. The software treats the `wire` as a Verilog-AMS `wreal` value. For example:

```
entity vh3 is
    port (vh3_port_net : IN real);
end;

module top;
    wire w;
    vh3 v1(w); // w coerced to wreal
endmodule
```

- You can connect VHDL bidirectional ports of type `real` to Verilog-AMS `wreals`. The resolution function used at the language boundaries is selected based on the drivers present. If there are no Verilog-AMS drivers present, the VHDL resolution function is used at the language boundary. If the Verilog-AMS drivers are present, the Verilog-AMS `wreal` resolution is selected to resolve the values at the mixed-language boundary. The resolution semantics on the signals that are not directly connected to the mixed-language boundary continue to follow the semantics of the language in which they are defined.
- You can connect a VHDL-AMS terminal port to a Verilog (digital) net or a VHDL (digital) signal port of type `std_logic` or `std_ulogic` to a Verilog-AMS analog net.
The software manages discipline resolution, driver-receiver segregation, and connect module insertion (at the Verilog boundary) automatically. Connect modules are always Verilog-AMS connect modules (VHDL-AMS does not provide for using connect modules). The software supports driver access functions by querying the digital ports of connect modules.
- You can connect a Verilog-AMS `wreal` net to a VHDL port of type `std_logic` when a Verilog-AMS `wreal` module is on top of a VHDL module. An R2L interface element is inserted between the `wreal` net and the `std_logic` port.

- You can connect a VHDL-AMS terminal or signal port to a Verilog-AMS analog or domainless net without using a shell.
- You can connect a VHDL (digital) `real` signal port to a Verilog-AMS `wreal` net without using a shell.
- You can connect a VHDL (digital) `real` vector to Verilog-AMS `wreal` array. Following are supported:
 - Verilog instantiation of VHDL with `wreal` array in port connection
 - VHDL instantiation of Verilog with `real` vector in port connection
 - Input/output ports of the above
 - Verilog-VHDL-Verilog and VHDL-Verilog-VHDL connections
 - VHDL-derived `real` vector connections
 - `wrealXState` and `wrealZState` for VHDL/mixed-language design
 - Reversed vector/array definitions ([1:0] vector/array definition connects to [0:1])

If there is a mismatch in the port width in any of the above scenarios, the simulator generates an error.

- You can make composite connections between VHDL (digital) signals and Verilog-AMS nets.
- Any VHDL connection to an SIE interface element must:
 - Correctly collapse the VHDL portion of the net, just as with normal port connections
 - Take any VHDL drivers into account in the shadow input
- For SIE interface element to VHDL connection:
 - Collapsed nets will include VHDL drivers
 - Resolved value of collapsed net will be fed back to VHDL
 - Shadow net will be fed by VHDL drivers converted to Verilog
- For a tran gate to VHDL connection:
 - Collapsed net, which is connected to tran network, will include converted VHDL drivers
 - Output of tran network will be fed to VHDL
 - Shadow net and tran network will include VHDL drivers

Importing Verilog-AMS Modules into VHDL

This technique for importing Verilog-AMS modules into VHDL requires the use of the Cadence® `library.cell.view` configurations, sometimes referred to as 5x configurations, for elaboration.

Verilog-AMS is a mixed-signal language and VHDL is digital only. If you want to import a Verilog-AMS module that has explicitly declared analog ports into VHDL:

1. Wrap the Verilog-AMS module so that it appears to the simulator to be a VHDL module. When you wrap a Verilog-AMS module, there are two shells: A digital Verilog shell wrapped in a VHDL shell.
2. Use the `xmshell` command to generate these shells.

If the Verilog-AMS module you want to import has `wire` or explicitly declared digital ports, you do not need shells because the elaborator can make the necessary connections. If you are uncertain as to whether you need shells or not, you can try elaborating without shells. If you need shells, the elaborator returns a message similar to the following:

```
xmelab: *E,AMSILC: Illegal port connection - :top:verilog_cell.v_a is an analog port
(line: 6, file: ./foo.v) and it cannot be connected directly to a VHDL digital signal
above.
```

Related Topics

- [xmshell Command](#)
- [Importing a Verilog-AMS Module](#)

Importing a Verilog-AMS Module

To import a Verilog-AMS module with explicitly declared analog ports into VHDL, do the following:

1. Use the `xmvlog` compiler to compile the Verilog-AMS source code for the module that you want to import.
2. Use the `xmshell` command to generate model import shells for the module you want to import.
3. In the source VHDL file, specify the architecture name to be used for entity.

If you specified the `-view` option for the `xmshell` utility in the previous step, use that name for the architecture. If you did not specify the `-view` option, use `verilog` for the architecture name.

4. Add a clause in the source VHDL file to specify the package.

If you specified the `-package` option for the `xmshell` command in step 2, use a clause such as
`use library. packagename. all;`

If you did not specify the `-package` option, use a clause such as
`use library. HDLModels.all;`

5. Compile all VHDL source code using `xmvhdl`.
6. Elaborate the design using the `xmelab` elaborator.

Example

To import the Verilog-AMS module in the file `comparator_analog.v`, do the following:

```
`include "discipline.vams"
`include "constants.vams"

module comparator(cout, inp, inm);
    output cout;
    input inp, inm;
    electrical cout, inp, inm; // The ports are explicitly analog
                               // so shelling is required.
    parameter real td = 1n, tr = 1n, tf = 1n;
    parameter integer i = 4.5, j = 5.49;
endmodule
```

1. Compile the Verilog-AMS source code.

```
xmvlog -ams -use5x comparator_analog.v
```

If your working library is `amsLib`, this command compiles the module `comparator` into `amslib.comparator:module`. The `-use5x` option is used because 5x configurations are used later to elaborate the design.

2. Generate model import shells using `xmshell`. The argument to the `xmshell` command is the `Library.Cell:View` specification for the compiled module.

```
xmshell -import verilog -ams -into vhd1 -generic amslib.comparator:module
```

This command generates two shells. The digital Verilog shell, generated in the file `comparator.vds`, looks like this:

```
module comparator(cout, inp, inm);
    output cout ;
    input inp ;
    input inm ;

    parameter td = 1.000000e09, tr = 1.000000e-09, tf = 1.000000e-09,
               i = 5, j = 5 ;

    comparator #(.td(td), .tr(tr), .tf(tf))
        (* integer view_binding="module"; *) comparator1
        (.cout(cout), .inp(inp), .inm(inm));
endmodule
```

This module has an instance `comparator1` of the same cell name `comparator`. It also has a view binding attribute `module`, which binds the instance to the `amslib.comparator:module` view. The source that is compiled into the `amslib.comparator:module` view is a Verilog-AMS description. This switches the elaborator into using the Verilog-AMS language.

The VHDL shell, generated in a file called `comparator.vhd`, looks like this:

```
library ieee;
use ieee.std_logic_1164.all;
entity comparator is
    generic (
        td: real := 1.000000e-09;
        tr: real := 1.000000e-09;
        tf: real := 1.000000e-09,
        i: integer := 5,
        j: integer := 5
    );
    port (
        cout: out std_logic;
        inp: in std_logic;
        inm: in std_logic
    );
end comparator;

architecture verilog of comparator is
    attribute foreign of verilog:architecture is "VERILOG(event)
    amslib.comparator:digital_shell";
begin
end;
```

Notice that the name of the architecture in the shell defaults to `verilog`.

The architecture `verilog` has a foreign attribute `amslib.comparator:digital_shell`, which tells the elaborator that the architecture is actually a shell for the Verilog module compiled into the view `amslib.comparator:digital_shell`. This attribute causes the elaborator to switch from the VHDL language into digital Verilog.

The `xmshell` utility also generates the following VHDL component declaration in the file `comparator_comp.vhd`:

```
library ieee;
use ieee.std_logic_1164.all;

package HDLModelsis

component comparator
generic (
    td: real := 1.000000e-09;
    tr: real := 1.000000e-09;
    tf: real := 1.000000e-09,
    i: integer := 5,
    j: integer := 5
);

port (
    cout: out std_logic;
    inp: in std_logic;
    inm: in std_logic
);
end component;

end HDLModelsis;
```

In Verilog-AMS, identifiers are case sensitive. By default, mixed-case and uppercase identifiers in Verilog-AMS are escaped in VHDL shells. For example, if the Verilog-AMS module is `vlog`, this identifier appears in the VHDL shell as `\Vlog\`. Use the `-noescape` option if you want the Verilog-AMS module name to be matched exactly in the shell. Do not set the `CDS_ALT_NMP` environment variable, which is not supported.

3. In the source VHDL file, specify that architecture `verilog` is to be used for

entity `comparator` (because this example uses the default value). Add the `use` clause (using the default version). With these changes, the source VHDL file for this example looks like:

```
-- top.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.HDLModels.all;
entity top is
end top;

architecture testbench of top is
    signal in1, in2, output : std_logic;
    for all: comparator use entity work.comparator(verilog);
begin

test: process begin
    in1 <= '0';
    in2 <= '0';
    wait;
end process;

comparator12: comparator port map(output, in1, in2);

end;
```

The `for all` statement binds the comparator instance to the design unit in `amslib.comparator:verilog`, which is a compiled version of the architecture `verilog`.

4. Compile the top-level VHDL file (`top.vhd`) and the VHDL package generated earlier by `xmshell`.

```
xmvhdl -v93 comparator_comp.vhd top.vhd -use5x
```

5. Elaborate the design with `xmelab`. Assuming that the corresponding 5x configuration is `amslib.top:config` and that the `connectrules` module is compiled into `amslib.AMSconnect:module`, you can elaborate the design with a command such as

```
xmelab amslib.top:config AMSconnect -discipline logic
```

In this example, using the `-discipline logic` option sets the discipline of the shell ports to logic.

Related Topics

- [Importing Verilog-AMS Modules into VHDL](#)
- [xmshell Command](#)

xmshell Command

The following section describes only those options required to import Verilog-AMS modules (that have explicitly declared analog ports) into VHDL modules. For detailed information about using the `xmshell` command, see "Generating a Shell with `xmshell`" in the "Mixed Verilog/VHDL Simulation" chapter of *Cadence VHDL Simulation User Guide*.

xmshell-import verilog -ams -into vhdl [`other_options`] `lib.cell:view`

<code>-import verilog</code>	Indicates that the imported model is Verilog, Verilog-A, or Verilog-AMS	
<code>-ams</code>	Indicates that the imported model is a Verilog-AMS (or Verilog-A) module	
<code>-into vhdl</code>	Indicates that you want to import the model into a VHDL module	
<code>other_options</code>	Zero or more of the following options or the options described in "xmshell Command Options" in the "Mixed Verilog/VHDL Simulation" chapter of the <i>Cadence VHDL Simulation User Guide</i> .	
	<code>- package name</code>	Enables you to specify the package name for the VHDL package that <code>xmshell</code> generates.
	<code>-view name</code>	Enables you to specify the cellview name to use for the digital Verilog shell and for the architecture name of the VHDL shell.
	<code>-generic</code>	If the imported Verilog-AMS module has parameters, you must use this option to make those parameters available in the shell.
<code>lib.cell:view</code>	Specifies the compiled design unit you want to import If you have not compiled the design unit, you can add the <code>-analyze</code> option to the <code>xmshell</code> command.	

For example, if `comparator` is a compiled Verilog-AMS module that uses parameters, you can create a shell for it using the following command:

```
xmshell -import verilog -ams -into vhdl -generic comparator
```

If `comparator` is not a compiled Verilog-AMS module, you can use the `-analyze` option as follows to

specify the source file that contains the `comparator` module:

```
xmshell -import verilog -ams -into vhdl -generic -analyze comparator.vams comparator
```

`xmshell` does not support the Verilog-AMS `wreal` port type.

Related Topics

- [wreal](#)
- [Importing a Verilog-AMS Module](#)
- [Importing Verilog-AMS Modules into VHDL](#)

Inherited Connections in VHDL-AMS

Using the AMS Designer simulator, you can define inherited connections in a VHDL-AMS entity and architecture (only on VHDL-AMS terminals) that you instantiate in a Verilog-AMS module. With inherited connections, you can inherit and use power and ground signals that you defined at a higher level of the hierarchy in a lower level of the design using net expressions. You can use these inherited connections instead of having to create explicit power and ground terminals at each level of the design hierarchy.

You can override VHDL-AMS inherited connections `netSet` properties in Verilog-AMS scope only. You cannot use `netSet` properties in VHDL-AMS scope. Also, you cannot use supply-sensitive attributes in VHDL-AMS scope. As a result, you cannot insert a supply-sensitive connect module at the boundary between Verilog-AMS and VHDL-AMS. Inherited connections in VHDL-AMS must be scalars only. You cannot define inherited connections that are VHDL arrays or electrical vectors.

In VHDL-AMS, you define inherited connections using `inh_conn_prop_name` and `inh_conn_def_value` attributes as follows:

```
package cds_inhconn_attr;
    attribute inh_conn_prop_name: string;
    attribute inh_conn_def_value: string;
end cds_inhconn_attr;

Use work.cds_inhconn_attr.all;

Entity inh_conn_cell is
    <code here>
End inh_conn_cell;

Architecture demo of inh_conn_cell is
    terminal vdd: electrical;
    attribute inh_conn_prop_name of vdd : terminal is "vddProp";
    attribute inh_conn_def_value of vdd : terminal is "cds_globals.\vdd!\ ";
Begin
    <code here>
End demo of inh_conn_cell
```

The `vdd` terminal inherits the `vddProp` property which has a default value of `cds_globals.\vdd!\` , where `cds_globals` is a Verilog module and `vdd!` is a global signal.

You can instantiate a VHDL-AMS entity in a Verilog-AMS module and use an explicit Verilog-AMS `cds_net_set` expression for an inherited connection as follows:

```
module top;

    electrical local_net;
    inh_conn_cell
        (* integer cds_net_set[0:0]="vddProp";
           integer vdd = "cds_globals.\ \ vdd5v! ";
        *)
    inst1();

    inh_conn_cell
        (* integer cds_net_set[0:0]="vddProp";
           integer vdd = "local_net"; // Overriding using a local net
        *)
    inst2();

Endmodule
```

Mapping Verilog-AMS Disciplines to VHDL-AMS Natures

The following table shows compatible mappings between Verilog-AMS disciplines and the predefined VHDL-AMS natures in the IEEE library and in the Cadence-provided VHDL-AMS variant of that library. The `disciplines.vams` file (that you normally include in your Verilog-AMS files) contains these discipline definitions. Using these mappings, you can connect Verilog-AMS components of one of the listed disciplines to a VHDL-AMS component of the corresponding nature.

Verilog-AMS Discipline	VHDL-AMS Nature
electrical	electrical
magnetic	magnetic
thermal	thermal
kinematic	translational
kinematic_v	translational_velocity
rotational	rotational
rotational_omega	rotational_velocity
None	fluidic
None	radiant

To create a mapping between a user-defined nature and a discipline, you can use the `MAPN2D` statement in your `hdl.var` file.

SPICE Built-In Primitives in VHDL-AMS and VHDL-Digital

Cadence provides a methodology to instantiate SPICE primitives in VHDL-Digital and VHDL-AMS using the following methods:

- [Component Instantiation of SPICE Primitives in VHDL-AMS](#)
- [Component Instantiation of SPICE Primitives in VHDL-Digital](#)
- [Direct Instantiation of SPICE Primitives in VHDL-AMS](#)

- [Instantiation of SPICE Primitives Through SPICE Models](#)

Component Instantiation of SPICE Primitives in VHDL-AMS

Cadence provides a package file containing predefined VHDL components corresponding to some of the common SPICE primitives. This file is located at:

```
${CDS_INST_DIR}/tools/affirma_ams/etc/vhdlams_spice_primitives/cds_spice_primitives.vhms
```

You can use the appropriate component from this file to instantiate the SPICE primitives you need. If none of the predefined components suits your needs, you can define your own VHDL component to instantiate the SPICE primitive.

Generic `cds_gm` is hardcoded to be used for general SPICE parameters with the exception of `real` vectors.

If you want to instantiate a `resistor` in VHDL-AMS, you could define a component and instantiate the component as follows. The tool will verify that the component name `resistor` is a SPICE primitive and the instance `r1` will be an instantiation of the SPICE primitive `resistor`.

```
ARCHITECTURE .....  
COMPONENT resistor IS  
  GENERIC (r : real);  
  PORT (TERMINAL x : ELECTRICAL;  
        TERMINAL y : ELECTRICAL);  
END COMPONENT;  
TERMINAL p1, p2 : ELECTRICAL;  
BEGIN  
  r1 : resistor GENERIC MAP (1000.0) PORT MAP (p1, p2);  
END;
```

If the name of the VHDL component defined does not match a SPICE primitive name, you can use the VHDL attribute `cds_spice_builtin` to rename the component, as shown in the following example. With this attribute, the instances of `myresistor` will become instantiations of the SPICE primitive `resistor`.

```
COMPONENT myresistor IS  
  GENERIC (r : real);  
  PORT (TERMINAL x : ELECTRICAL;  
        TERMINAL y : ELECTRICAL);  
END COMPONENT;  
attribute cds_spice_builtin of myresistor : component is "resistor";
```

Component Instantiation of SPICE Primitives in VHDL-Digital

When VHDL digital signals connect to SPICE primitives, you need to create your own components for the SPICE primitives and instantiate the components. In this scenario, CEs will be required and need to be precompiled, which will be automatically inserted by the tool.

For example, if you want to instantiate a resistor in VHDL-Digital, you could define a component and instantiate the component as shown in the following example. The tool will insert the CE automatically for `p1` and `p2`. The instance `r1` will be an instantiation of the `resistor` SPICE primitive.

```
COMPONENT resistor IS
    GENERIC (r : real);
    PORT (SIGNAL x : out STD_LOGIC;
          SIGNAL y : in STD_LOGIC);
END COMPONENT;
SIGNAL p1, p2 : STD_LOGIC;
begin
    r1 : resistor GENERIC MAP (1000.0) PORT MAP (p1, p2);
ends;
```

Direct Instantiation of SPICE Primitives in VHDL-AMS

Direct instantiation of SPICE primitives is supported through the use of a Cadence-provided file that contains some predefined common skeletons for built-in primitives. This file is located at:

```
${CDS_INST_DIR}/tools/affirma_ams/etc/vhdlams_spice_primitives/cds_spice_primitives_entities.vhms
```

In the skeletons, parameters such as port type, generic type, name, etc. are hardcoded. The predefined file needs to be precompiled also. If none of the predefined skeletons in this file suit your requirement, use component instantiation.

Instantiation of SPICE Primitives through SPICE Models

You can instantiate SPICE models as SPICE built-in primitives by specifying the model file with `MODELPATH` option of the `xrun` command, as shown in the following example.

```
xrun -modelpath model.ckt .....
```

model.ckt:

```
simulator lang = spectre
model myresistor resistor r=1 l=2u w=2u
```

SPICE model instantiation:

```
component myresistor is -- model of resistor, defined in model.ckt
    port (terminal a, b : electrical);
end component;

r2 : myresistor port map (a, b);
```

You can also specify model files for an analog device using the `include` statement in the AMS control file.

String Type Literals and Generics in VHDL-AMS

You can use string type literals and generics in VHDL-AMS even if the objects are referenced in the analog context. For example:

```
if (rfunc = "cont") use
-- Continuous linear resistance change during transition

r_eff == r_sig'ramp(tdbrk, tdmk);

else

-- Discontinuous resistance change during transition

r_eff == r_sig;

end use;
```

In the above example, `rfunc` is a string type generic.

Currently, the AMS Designer simulator supports only built-in string types (defined in the STD package). The following are not supported:

- String type signals, user-defined functions with string type arguments, user-defined functions returning a string type, or string type variables (referenced in the analog context)
- Concatenation operator on string types used in the analog context
- The `fred` and `succ` operators on string types used in the analog context.

VHDL-SPICE Conversion Element Optimization

The VHDL-D and SPICE blocks in a design are connected together using a Conversion Element (CE). CE optimization is a process to ensure that minimum number of CEs are required for a given hierarchical or local net that connects one or more SPICE blocks in a design. All unnecessary CEs in the design are eliminated to optimize the design performance.

The VHDL-SPICE CE optimization is performed in the scenarios discussed below as follows:

- **A single SPICE block is connected to a VHDL-D signal that is not read by or written to by any other VHDL-D signal**

In such a scenario, remove the CE from the SPICE block. The connected VHDL-D signal is treated as analog for computation purpose.

- **Two or more SPICE blocks are connected to each other through a VHDL-D net (either hierarchical or local) that is not read by or written to by any other VHDL-D signal**

In such a scenario, remove the CEs from all the SPICE blocks. The VHDL-D interconnect is treated as analog for computation purpose.

- **Two or more SPICE blocks are connected to each other through a VHDL-D net (either hierarchical or local) that is read by or written to by other VHDL-D signals**

In such a scenario, consider the following cases for each net connection:

- If the direction of all SPICE ports connected to the VHDL-D net is the same, keep any one of the CEs and ignore all others. This ensures that there is only one CE between all the SPICE blocks and the VHDL-D signals.
- If the directions of the SPICE ports connected to the VHDL-D net are different, then traverse through all the VHDL-D signals connected to the VHDL-D net and perform one of the following actions:
 - If all VHDL-D signals have the direction `IN`, keep only one `A2D` CE and ignore all others.
 - If all VHDL-D signals have the direction `OUT`, keep only one `D2A` CE and ignore all others.
 - If VHDL-D signals are all `INOUTs` or have different directions, an error message is displayed.

Related Topic

- [CE Information in VHDL-SPICE](#)

CE Information in VHDL-SPICE

In AMS Designer simulator, you can use the elaborator option `-cereport` to generate a report with information about the final set of CEs inserted in a VHDL-SPICE design.

Only the CEs specified in the AMS control block `ce` card are identified and reported by this option. The report contains the following information about the CEs:

- CE serial number
- Name of the CE (complete name containing library entity and architecture)
- VHDL file name from where the CE is picked up
- Hierarchical name of the CE instance
- Generic map used for the `ce` instance
- VHDL-Digital signal type and the spice cell or port name associated with the CE

The report also clearly distinguishes between the optimized CEs and the CEs that are not optimized.

In addition to `-cereport`, you can use the following elaborator options to generate specific CE information:

- `-ceverbose` - To generate detailed VHDL-SPICE CE report for debugging the issues related to CE insertions. In addition to the information provided by the `-cereport` option, the `-ceverbose` option reports the following.
 - Information about the digital drivers for L2E
 - Information about the digital load for E2L
 - Back-to-back information for an L2E, when the L2E driver is an E2L CE
 - Information about the drivers and loads for digital BIDIR input
- `-ceprobes` - To generate a Tcl file, `ceprobes.tcl`, with the probes for CE verification
- `-cedriversload` - To generate a Tcl file, `cedriversloads.tcl`, with the drivers and loads for the CE

The `-ceprobes` and `-cedriversload` options only work in conjunction with the `-cereport` or `-ceverbose` option. They automatically enforce `-cereport` if none of `-cereport` or `-ceverbose` is explicitly specified.

Related Topic

- [VHDL-SPICE Conversion Element Optimization](#)

Real Number Modeling

You can create designs using real number models ([RNM](#)) and simulate them using the AMS Designer simulator. Using either Verilog-AMS or VHDL-AMS languages, you can define real ports that are not `electrical`. You can use `wreal` connect modules to connect real number models to Spectre or SPICE or `electrical` blocks. The elaborator automatically inserts appropriate connect modules between `electrical` ports (analog domain) and `wreal` ports (digital domain).

You can use the `-iereport` or `-ieinfo` command-line options to generate a report summary for each type of connect module the elaborator inserts.

Real connect modules support analog/digital (discrete `wreal`) conversion with a custom variable rate, while `wreal` connect modules support variable rate signal conversion. The `absdelta` function drives the real connect module analog-to-digital conversion, while digital events drive the real connect module digital-to-analog conversion.

Basic `wreal` Features of the AMS Designer Simulator

Basic `wreal` features of the AMS Designer simulator consist of the following:

- Scalar `wreal` declaration
- Local `wreal` declaration in a behavioral construct
- Hierarchical connection of nets where you explicitly assign the `wreal` data type

You can use these features to create real-numbered models.

For example, the following Verilog-AMS model for a DC source has an output port type that is `wreal`, which is like a wire with the ability to transfer a real number:

```
module rdcsource ( aout );
output aout;
wreal aout;      //output wreal
\logic aout;     //discrete domain

parameter real dc=0.0;
real aoutreg;

initial begin
    aoutreg=dc;
end
assign aout = aoutreg;

endmodule
```

The following Verilog-AMS model for a ramp source has an external clock that defines the sampling rate:

```
module step (clk, y );
input clk;
output y;
wreal y;
\logic y;

parameter real offset=0.0;
parameter real step=1;

real yval;

initial
    yval=offset;

always @(posedge clk)
begin
    yval = yval + step ;
end

assign y = yval;

endmodule
```

Advanced wreal Features of the AMS Designer Simulator

Advanced `wreal` features of the AMS Designer simulator consist of the following:

- `wreal` arrays

For example:

```
module wreal_src_bus ( out );
```

```
output out[1:0];
wreal out[1:0];

assign out[0] = 3.2;
assign out[1] = 2.2;
endmodule
```

- `wreal` nets with more than one driver.
- `wrealXState` and `wrealZState`
- Connections between `wreal` and VHDL `real` signals
See also [Connecting VHDL and VHDL-AMS Blocks to Verilog and Verilog-AMS Blocks](#).
- `wreal` coercion for hierarchical connections between `wreal` and wire nets, `wreal` and SystemVerilog real nets, `wreal` and VHDL real nets

`wreal` coercion refers to the ability to connect `wreal` nets hierarchically to wires and coercing the wires to become `wreals`. This also includes support for hierarchical connection of a `wreal` net to SV-real and VHDL-real signals.

For example, the following discrete `wreal` step generator connects to an electrical resistor, so the elaborator coerces wire `aout` to `wreal` and inserts a bidirectional electrical -to-real connect

```
module: module top ( );
electrical gnd;
ground gnd;
wire aout;
\logic aout;    //discrete domain

    clock_gen #(.period(1.0)) I0 ( .clk( clk ) );
    step #(.step(10.0e-6)) I2 ( .clk( clk ), .y( aout ) );
    resistor #(.r(200.0)) RLOAD2 (aout, gnd);
endmodule
```

where module `step` might look like this:

```
module step (clk, y );
input clk;
output y;
wreal y;
\logic y;
```

```
parameter real offset=0.0;
parameter real step=1;
real yval;
initial
    yval=offset;
always @(posedge clk)
begin
    yval = yval + step ;
endassign y = yval;

endmodule
```

You can use the `xrun/xmelab` option `-rnm_coerce` with parameters to turn the `wreal` coercion feature `ON` or `OFF`. The default is `ON` for whole design.

See also [Instantiating VHDL Blocks with Real Signal Ports on a Schematic](#) and [Connecting VHDL and VHDL-AMS Blocks to Verilog and Verilog-AMS Blocks](#).

- `wreal` nets with inherited connections

This includes:

- Connection between a `wreal` nettype and another `wreal` nettype through an inherited connection.
- Connection between a `wreal` nettype and an `electrical` nettype through an inherited connection.

`Wreal` nets with inherited connections are also supported in the analog design environment ([ADE](#)) flow. However, for this, you need to explicitly define the target nets of the inherited connection to `wreal` nets.

Connection between the `wreal` nettype and the `real` nettype is not supported.

Example (`wreal` nettype with another `wreal` nettype):

```
module top;

wreal (* integer inh_conn_prop_name="global_prop_wreal";
      integer inh_conn_def_value="cds_globals.wr"; *) local;

always @(local) begin
$display("===== local=", local);
end

initial begin

#1 $display("===== (ini) local=", local);
    #1 $display("===== (ini) local=", local);
    #1 $display("===== (ini) local=", local);
end

endmodule
```

where module `cds_globals` might look like the following:

```
module cds_globals;

electrical el, gnd;
    wreal wr;
    real re;
    wire wi;
    reg rg;
    ground gnd;

    assign wr = re;
    assign wi = rg;

    vsource #(.dc(3.3)) v1 (el, gnd);

    initial begin
#0 re = 0.0; rg = 0;
    #1 re = 1.1; rg = 1;
    #1 re = 2.2; rg = 0;
    #1 re = 3.3; rg = 1;
    #1 re = 4.4; rg = 0;
    #1 re = 5.5; rg = 1;
end

endmodule
```

- `-wreal_resolution` command-line option for specifying a resolution function for `wreal` nets with more than one driver
- `wreal` as an independent variable in the [\\$table_model](#) function

For more information, see [Using wreal Independent Variables in a \\$table_model](#).

- Delays on `wreal` nets

Single net delays and assignment delays are supported on `wreal` nets. Assignment delays are supported when they are used with `wreal` nets that are part of continuous assignment declaration or net assignment declaration. This includes:

- real to `wreal` assignments

```
real r,  
wreal w;  
assign #5 w = r;
```

- `wreal` to `wreal` assignments

```
wreal x,  
wreal w;  
assign #5 w = x;
```

Following are not supported:

- Multiple delays in a net delay. For example:

```
wreal #(10,11) w;
```

- Multiple delays (rise, fall or rise, fall to-z) in an assignment delay. For example:

```
assign #(2,3,4) a = b;
```

Related Topics

- [Using the wreal Data Type](#)
- [Arrays of Real Nets](#)
- [Real Nets with More than One Driver](#)
- [Selecting a wreal Resolution Function](#)

Using the wreal Data Type

The AMS Designer simulator supports the Verilog®-AMS `wreal` data type, which lets you define a real number physical connection between structural entities.

```
wreal in; // Declares 'in' as a wreal net
```

A set of `wreal` sample libraries are installed under `INST_DIR/tools/amsd/wrealSamples` area. These

libraries contain text models and also the versions that can be used with the current Xcelium and Spectre releases. The text models are directly accessible and usable in a command-line digital-centric flow.

Related Topics

- [Basic wreal Features of the AMS Designer Simulator](#)
- [Advanced wreal Features of the AMS Designer Simulator](#)
- [Using Real Number Modeling in SystemVerilog](#)
- [Data Types and Objects](#)

wreal Independent Variables in a \$table_model

You can use the Verilog-AMS [\\$table_model](#) function in the analog context (in an `analog` block) and in the digital context (such as in an `initial` or an `always` block) with `wreal` independent variables.

Using the `$table_model` function in your functional and timing simulation, you can replace transistor circuitry with a table model, where the table model operates on real number sweep data that you measure. For example, consider the following Verilog-AMS module, which measures input and output voltage values and saves the data into an ASCII text file as a two-dimensional table that we can reuse in a `$table_model` function:

```
`include "disciplines.vams"
`timescale 1ns/100ps
module probe2table ( a, b, clk );
input a, b, clk;
electrical a, b;
parameter lsb=1u;
real previous_b_value=0;
integer fptr;
integer index=1;

initial begin
    fptr = $fopen("table2d.dat", "w");
    if( !fptr ) $stop;
    $display(" >> Open the file for writing" );
    previous_b_value = V(b);
end

always @( posedge(clk)) begin
    if (V(b) - previous_b_value > lsb ) begin
        $fwrite(fptr, "%d %1.9f %1.9f\n", index, V(a), V(b) );
        $display(">> Write the data t=%-6d ns V(a)=%1.9f V(b)=%1.9f ", $time, V(a), V(b));
        previous_b_value = V(b);
        index = index +1;
    end
end
endmodule
```

You can reuse the table model in the resulting `table2d.dat` file in a `$table_model` function that has a `wreal` as the independent variable as follows:

```
`define TABLE_FILE_NAME "table2d.dat"
`timescale 1ns / 100ps
module inv_table ( a, y );
    output  y;
    wreal   y;
    parameter real td=10p;

    input   a;
    wreal   a;
    real    y_reg;
    real    delay_ns= td / 1.0e9;

    initial begin
        y_reg = $table_model(a, `TABLE_FILE_NAME, "I,1L");

    end

    always @(a) begin
        #delay_ns y_reg = $table_model(a, `TABLE_FILE_NAME, "I,1L");
    end
    assign y = y_reg;

endmodule
```

In the `initial` and `always` blocks, you assign the interpolated data to a real variable, `y_reg`. The value of the `td` (real) parameter--which represents the propagation delay of an inverter gate--determines when to assign the value of `y_reg` to the `wreal` output value, `y`.

The software interpolates and extrapolates, using the control string you specify, to estimate each value from the known set of values. The first part of the control string (in this case, `I`) controls the numerical aspects of the interpolation process. For this example, the `I` causes the software to ignore the corresponding dimension (column) in the data file. The number (`1`) is the degree of the interpolation splines. The last part of the control string (in this case, `L`) specifies the extrapolation method: how the simulator evaluates a point that is outside the region of sample points included in the data file. The letter `L` indicates *linear* extrapolation.

You can use an environment variable to specify the location of the table file in the `$table_model` function in digital context (such as `initial` or `always` block). For example:

```
initial begin

out1 = $table_model(wb, "$WORKAREA/$FILENAME", "");

end
```

In addition, you can use real array as a table model data source in a digital block. For example:

```
real tvar1[0:50];

real ftyp[0:50];

always @ (en) begin

test_out_real = $table_model(w_t_in , tvar1, ftyp,"3CC");

end
```

Related Topic

- [Interpolating with Table Models](#)

Verilog-AMS `wreal` and Analog Signal Connections

In mixed-signal designs, you can connect `wreal` nets to `electrical` nets, with either one on top, provided that appropriate connect modules are available to translate between signal types. For example, you can specify the following connection (`wreal` connected to `electrical`) when you have appropriate connect rules and modules available.

```
module top;
    wreal w;
    child c1(w); // wreal connected to electrical
endmodule

module child(e);
    input e;
    electrical e;
endmodule
```

When the `wreal` and `electrical` nets are connected, the port can be `input`, `output`, or `inout`.

The elaborator uses the signal type (`wreal` or `electrical`) to identify appropriate connect rules. For

example, if `myrule` is defined as

```
connectrules myrule;
    connect a2d input electrical output logic;
    connect wreal_a2d input electrical output logic;
endconnectrules;
```

the domains and directions are insufficient to distinguish between the `a2d` rule and the `wreal_a2d` rule. If you add the information that the logic domain signal is actually `wreal`, the elaborator can select the connect module that has a `wreal` port of the appropriate direction.

You can find the [E2R](#) and [R2E](#) connect modules in the Cadence hierarchy:

`your_install_dir/tools/affirma_ams/etc/connect_lib/E2R.vams`

and

`your_install_dir/tools/affirma_ams/etc/connect_lib/R2E.vams`

These connect modules have appropriate port types and directions for use in connecting `wreal` nets to `electrical` nets. For example,

```
connectrules ConnRules_full_fast;
    connect L2E_2;
    connect E2L_2;
    connect Bidir_2;
    connect E2R;
    connect R2E;
    connect ER_bidir;
endconnectrules
```

In addition to connecting Verilog-AMS wreals to electrical nets, you can connect the SystemVerilog real ports to electrical nets.

The basic, full, and full_fast R2E connect modules have been enhanced to support the current mode, in which electrical current is converted between `wreal` and electrical connection.

Following new parameters have been added to these connect modules to support `wreal` current. You can specify all the parameters in the `ie` card.

Parameter Name	Default Value	Description
currentmode	0	1 for <code>wreal</code> current

clamp	0	When set to 1, enables voltage clamp. It limits the signal voltage level to be within the supply voltage range, regardless of the value of the wreal current. • With parameter <code>dvclamp</code>, the clamp zone is defined to be from <code>vss</code> to <code>vss + dvclamp</code> and from <code>vdd - dvclamp</code> to <code>vdd</code>. • When wreal current signal drives the electrical signal voltage to be entering clamp zone, the electrical current converted from the wreal signal is reduced so that the electrical voltage never goes beyond the supply voltage range. • When the signals goes into the clamp zone, the converted electrical signal is distorted and is not equal to wreal signal value. • When <code>clamp</code> is enabled, the simulation may be more stable. However, to ensure the accuracy of circuit function, you need to ensure that the converted electrical signal does not go into the clamp zone.
rin	200	Current load resistance
idelta	10u	Current delta of absdelta
itoldelta	<code>(`idelta) / 4</code>	Current tolerance of absdelta
ix	0	Current of wreal x
vbias	<code>`vss + (`vsup) / 2</code>	Bias voltage of current load
vcclamp	<code>(`vsup) / 20</code>	Voltage clamp zone

You can create a customized connect rule `CR_full_fast_current` with discipline `wreal_current` and define `wreal_current` on applicable wreal ports of Verilog-AMS modules.

The current polarity of wreal port for wreal current modeling should be defined such that:

- For wreal input port, positive current means current flowing into the module.
- For wreal output port, positive current means current flowing out of the module.

Note that this wreal port current definition is different than the port current definition of Verilog-A/SPICE, in which positive current means current flowing into the module/subckt (regardless the port direction declaration if any).

With this wreal port current definition and the Verilog-A port current definition, the Electrical-Wreal connect modules in current mode are basically modeled as:

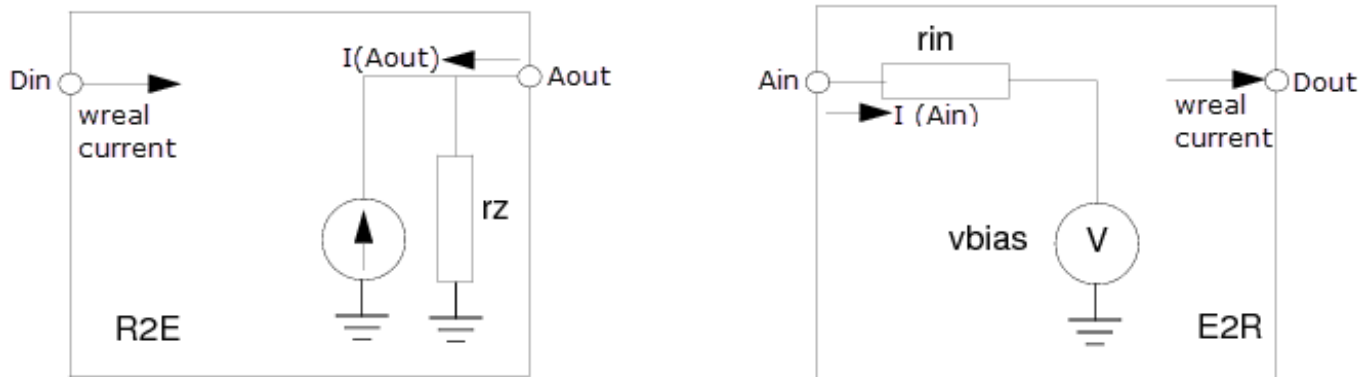
- R2E (input wreal Din, electrical Aout):

```
I(Aout) <- -Din;
```

- E2R (electrical Ain, output wreal Dout):

```
Dout = I(Ain);
```

The following figures show the equivalent circuits for current-based connect modules:



Related Topic

- [Resolving Disciplines for Verilog-AMS wreal Nets](#)

Resolving Disciplines for Verilog-AMS wreal Nets

The `elaborator` treats `wreal` nets as Verilog-AMS wires from a discipline resolution point of view. The `elaborator` resolves `wreal` nets to either the digital or the analog domain based on how they are connected in the design and how the discipline resolution process propagates domain and disciplines up and down the design hierarchy.

Like Verilog-AMS wires, `wreal` nets can have any discipline (discrete or continuous) based on how they are declared, used, or resolved. To keep `wreal` signals in the discrete domain, use the `-ams_dig_wreal` option

Using default discipline resolution with a default discipline of `logic`, discrete nets that connect to `wreal` nets can become `wreal` nets. Consider the following example:


```
module top;
  wreal a;
  child1 c1(a);
endmodule

module child1(b) ;
  input b;
  logic b;
  child2 c2(b);
endmodule

module child2(c);
  input c;
  electrical c;
  child3 c3(c);
endmodule

module child3(d);
  input d;
  wire d;
  child4 c4(d);
endmodule

module child4(e);
  input e;
  wire e;
  assign e = 1'b1;
  ...
endmodule
```

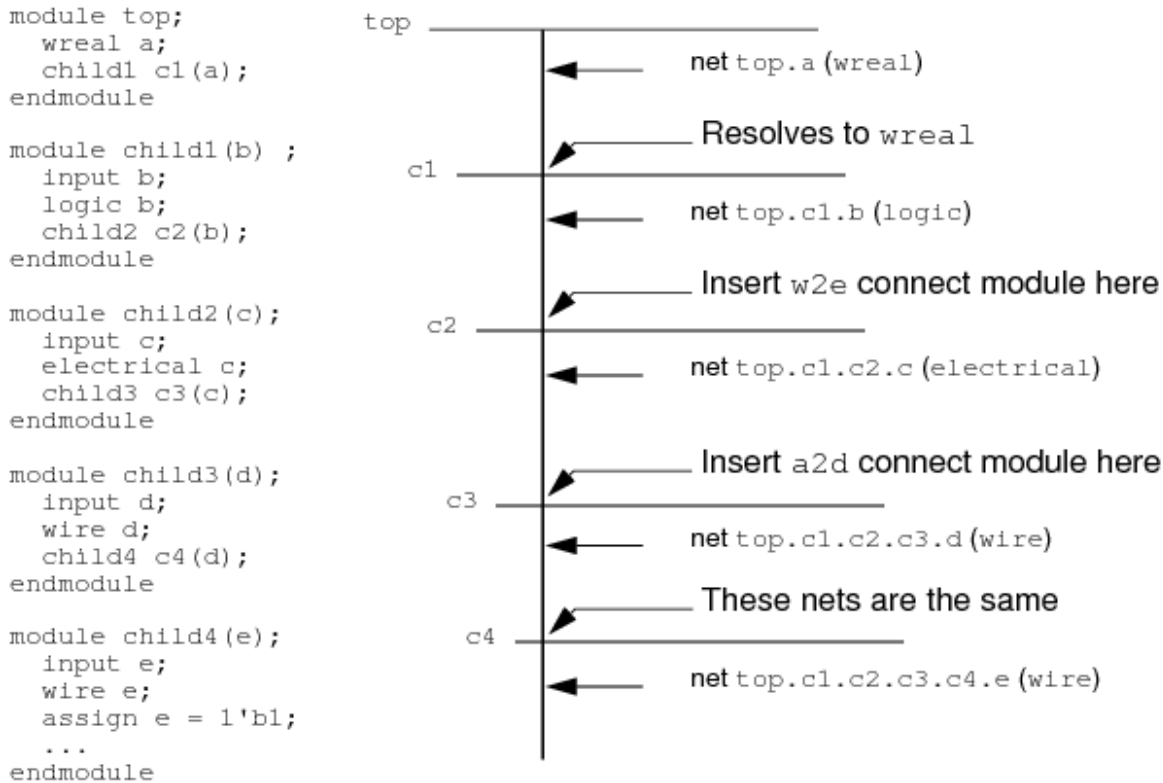
According to the default discipline resolution where the default discipline is `logic`, nets `top.a` (a `wreal`) and `top.c1.b` (discrete net with discipline `logic`) are equivalent and `top.c1.b` resolves to a `wreal` (discipline `logic`).

Nets `top.c1.b` (discrete) and `top.c1.c2.c` (continuous) have incompatible disciplines (`logic/wreal` versus `electrical`) as do nets `top.c1.c2.c` and `top.c1.c2.c3.d` (`electrical` versus `logic/wire`) so the software needs to insert connect modules where these nets connect. The software determines which connect module to insert using connect rules. For example:

```
connectrules myrules;
  connect w2e input logic, output electrical;
  connect a2d input electrical, output logic;
endconnectrules
```

The port value type (`wreal`, `electrical`, `logic`) defines the selection of a connect module. The first connect rule above says to use the `w2e` connect module when connecting `wreal` (defaults to `logic` discipline) to `electrical` nets. The second connect rule above says to use the `a2d` connect module when connecting `electrical` to `logic` nets.

Nets `top.c1.c2.c3.d` and `top.c1.c1.c3.c4.e` are both discrete wires so the software does not need to insert connect modules here.



In the absence of unidirectional ports, the number of drivers on the `wreal` net determines the behavior of the connect module. For example, when an `inout wreal` connects to an `inout electrical`, you can use the `$driver_count` function in the connect module definition to determine the number of drivers on the `wreal` net. If there are no drivers on the `wreal` net, the connect module does the conversion from `electrical` to `wreal`. (See [Verilog-AMS wreal and Analog Signal Connections](#))

VHDL Blocks with Real Signal Ports on a Schematic

When you instantiate a VHDL block containing real signal ports on a schematic, you create a connection of VHDL real signals to Verilog-AMS wires from above. In such a situation, the simulator coerces the Verilog-AMS wire to become a digital `wreal` to integrate such VHDL blocks into the AMS flow.

AMS Designer resolves Verilog-AMS `wreal` nets to the analog or digital domain during the discipline resolution process. AMS Designer does something similar with Verilog-AMS wires. When AMS Designer resolves to the digital domain, it is the digital simulator that handles the computations for that net.

Related Topic

- [Verilog-AMS `wreal` and Analog Signal Connections](#)

wreal Nets at Mixed-Language Boundaries

You can use `wreal` nets at mixed-language boundaries. However, if you have any Verilog-AMS `wreal` nets driving a VHDL real number receiver, that receiver must be able to handle the `wrealZState` and `wrealXState` state values. The software represents these values as double-precision (64-bit) NaNs (Not-a-Number) using the IEEE 754 Standard for Binary Floating-Point Arithmetic. The `wrealZState` value has a 64-bit pattern of 0xFFFFFFFF00000000 and the `wrealXState` value is any NaN that does not have the bit pattern of `wrealZState`. The set of connect rules that Cadence provides (in `your_install_dir/tools/affirma_ams/etc/connect_lib`) supports `wrealZState` and `wrealXState` state values.

See [Selecting a `wreal` Resolution Function](#) for information about using the `wreal_resolution` command-line option during elaboration to select a `wreal` resolution function.

The set of connect rules that Cadence provides is available at, `your_install_dir/tools/affirma_ams/etc/connect_lib/README`.

Related Topic

- [Resolving Disciplines for Verilog-AMS `wreal` Nets](#)

Writing Bidirectional Model Behavior

You can use the `$cds_get_external_drivers` system task in module definitions to model bidirectional behavior like pass gates, for nets with user-defined nettypes (UDNs). For example, you may use `$cds_get_external_drivers` in module definitions to model the bidirectional behavior of WTRAN models with SV-`wreal` nettypes.

You can also use the abbreviated name, `$cged` instead of `$cds_get_external_drivers`.

The use model is similar to the connect modules that support SIE with the following differences:

- The module cannot contain any analog ports
- The module is not limited to just two ports like the SIE connect modules

The module can have an arbitrary number of digital ports. All inout ports that have a `$cds_get_external_drivers` call associated with them receive additional SIE processing to create shadow networks. All ports that do not have a `$cds_get_external_drivers` call associated with them receive normal SystemVerilog port processing.

The module cannot have a `$cds_get_external_drivers` call associated with a port that is not declared (or has default direction) as `inout`. Also, all ports that have

a `$cds_get_external_drivers` calls associated with them must be nets;

`$cds_get_external_drivers` cannot be called on a var port. However, normal ports that do not get SIE processing can have any legal port kind, type, or direction. Also, module/UDP instantiation inside SIE model is not supported.

Starting with Xcelium 19.09, bidirectional behavior of WTRAN models is supported with mixed-language (Verilog/VHDL) net connections also. So, you can instantiate Verilog SIE model inside VHDL (VHDL on parent and Verilog SIE model on child).

Following is an example of a `wrealldriver` pass gate model that can be modeled with SIE ports:

```
import cdn_rnm_pkg::*;
module wrealldriver_pass_gate(inout wrealldriver a, inout wrealldriver b, wire en);
real aVal, bVal;
real aOut, bOut;
initial
begin
$cds_get_external_drivers(a, aVal);
$cds_get_external_drivers(b, bVal);
aOut = `wrealZState;
bOut = `wrealZState;
end

always @(aVal, bVal, en)
begin
//en is high so the pass gate is open
if(en===1)
begin
if(aVal === `wrealZState)
begin
// a is Z so we pass b to a
aOut = bVal;
bOut = `wrealZState;
end
else if(bVal === `wrealZState)
begin
// b is Z so pass a to b
bOut = aVal;
aOut = `wrealZState;
end
else
begin
// both a and b are nonZ so we have contention and output X
aOut = `wrealXState;
bOut = `wrealXState;
end
end
else if(en===0)
begin
// en is low so we aren't passing values. just drive Z
aOut = `wrealZState;
bOut = `wrealZState;
end
else
begin
// en is X or Z so output X
aOut = `wrealXState;
bOut = `wrealXState;
end
end
// assign the pass values to the port so they are visible to external receivers
assign a = aOut;
assign b = bOut;
endmodule
```

In the above example, if the enable is high and only one of the model ports is driven, that value is passed to the other side. For cases where both ports are driven and the enable is high, both sides of the pass gate resolve to `wrealXState`. If the enable is low, both sides of the pass gate are isolated from each other (the pass gate drives out ``wrealZValue`). If the enable is X or Z, both sides resolve to ``wrealXState`.

Following is an example of a `WTRANIF1` model that can be modeled with SIE ports:

```
import cds_rnm_pkg::*;

module wtranif1(inout wreal4state a, inout wreal4state b, input var bit en);
  real a_val;
  real b_val;
  wreal4state a_ord;
  wreal4state b_ord;
  real Zval;
  real Xval;

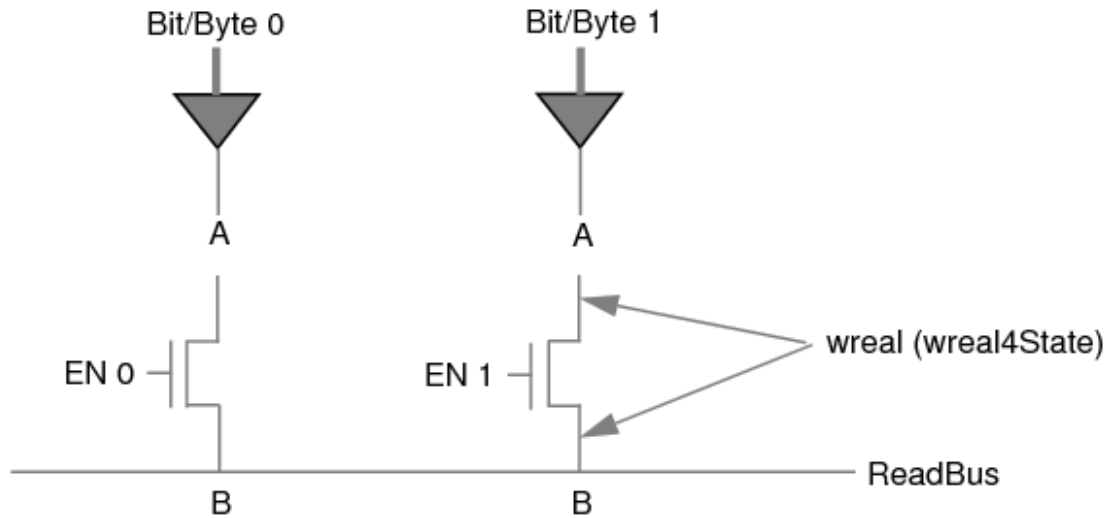
  initial
  begin
    Zval = `wrealZState;
    Xval = `wrealXState;
    $cds_get_external_value(a, a_ord);
    $cds_get_external_value(b, b_ord);
  end
  /* disable full resolution on the shadow net */
  assign a_ord = (en!=1)? Xval : Zval;
  assign b_ord = (en!=1)? Xval : Zval;

  assign #1 a = (en!=1'b1)? Zval : b_ord;
  assign #1 b = (en!=1'b1)? Zval : a_ord;

endmodule // wtranif1
```

`wreal4state` is analogous to a digital 4-state wire and is the only type that can be used to model a `WTRANIF` gate.

The following figure illustrates a typical use scenario of the `WTRANIF1` switch.



The figure illustrates two memory cells (Bit/Byte X) sharing a common read bus. The wtranif1 switch acts like a sense amplifier that drives the read bus with the content of the memory cells, when enabled individually (EN X).

The `$cds_get_external_value` function is used to resolve the drive strength of A and B.

- When enable EN=1
 - If A is `wrealZState` and B is not, the value of B drives A
 - If B is `wrealZState` and A is not, the value of A drives B
 - If none of them are `wrealZState`, they resolve to `wrealXState`
- When enable EN=0, A and B maintain their values

Modeling bidirectional behavior for WTRAN models may affect elaboration performance. You can use the `xrun` option, `-amsie_task_perf`, to optimize elaboration performance for such models.

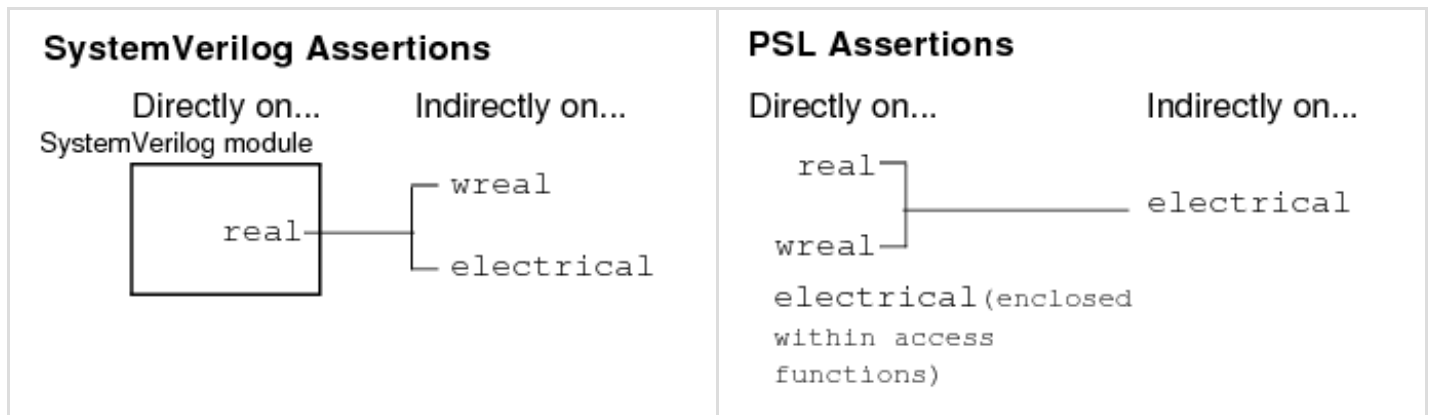
Modeling arbitrary networks of pass gates may not be possible for every resolution function as the pass gate model relies heavily on the resolution behavior of the net. For some resolutions (like for instance `wrealavg`), it may not be possible to use this methodology to model pass gate like behavior (`wrealavg` ports may have different values on each port because the internal driver gets averaged with the external drivers, which may be different on each port in a multi-driver situation). Therefore, if you intend to model pass gate behavior, you should define the resolution function such that it supports your pass gate models.

Related Topics

- [Resolving Disciplines for Verilog-AMS wreal Nets](#)
- [wreal Nets at Mixed-Language Boundaries](#)

Using wreal in Assertions

The AMS Designer simulator lets you use SystemVerilog Assertions (SVAs) on SystemVerilog `real` variables (or ports) that connect to `wreal` or `electrical` nets, and PSL assertions on `real` and Verilog-AMS `wreal` nets that connect to `electrical` nets.



Related Topic

- [SystemVerilog Assertions on real, wreal, and electrical Nets](#)

Selecting a wreal Resolution Function

You can define a resolution function for each net in a design on a per scope basis by using the Verilog-AMS discipline mechanism. You can also designate a global resolution function that can be used for undisciplined nets and for the nets with disciplines that do not define a `wreal` resolution function.

You can define resolution function in the following ways:

- **Defining a wreal Resolution Function for a Discipline.** You can define `wreal` resolution functions for a discipline in the following two methods:
 - **Defining a Resolution Function Directly in the Discipline Definition.** To define a `wreal` resolution function for a discipline, use the `realresolve` parameter in the

discipline definition. The syntax of this parameter is as follows:

```
realresolve resolveFunction
```

where the `resolveFunction` be any of the predefined wreal resolution functions such as `default`, `fourstate`, `sum`, `avg`, `min`, `max`, `none`. See [Predefined wreal Resolution Functions](#).

Note that wreal resolution function names are case-sensitive. Therefore, if you specify the `sum` resolution function name as `Sum` or `SUM`, it will result in an error.

If you specify the value of `resolveFunction` as `none`, it is equivalent to not using the `realresolve` parameter at all. Nets of such disciplines use the global resolution function. An error occurs if an invalid value is specified for the `resolveFunction` variable.

The following example defines a discrete discipline named `wrealavg` that uses the `avg` resolution function for wreal nets.

```
discipline wrealavg
    domain discrete;

    realresolve avg;

enddiscipline
```

- **Defining a Resolution Function with the AMSCB connectmap Card.** You can use the `realresolve` parameter with the `connectmap` card to define a resolution function for one or more disciplines. See [connectmap](#).
- **Defining a Global wreal Resolution Function.** You define a global `wreal` resolution function in the following two methods:

- **Defining Global wreal Resolution Function Using `-wreal_resolution`.** You can use the `-wreal_resolution` command-line option with `xmelab` or `xrun` to specify how you want the elaborator to resolve `wreal` nets that have more than one driver. The format of this command-line option is as follows:

```
-wreal_resolution resolutionFunction
```

where `resolutionFunction` can be any of the predefined wreal resolution functions such as `default`, `fourstate`, `sum`, `avg`, `min`, `max`, `none`. See [Predefined wreal Resolution Functions](#).

If you do not specify a resolution function using this command-line option, the elaborator uses the `default` resolution function algorithm.

For those cases where the elaborator issues a run-time warning message, you can cause the message to be fatal using the `-xmfatal` command-line option. For more information about this option, see [-xmfatal](#).

 The `none` resolution function is not a valid option to `-wreal_resolution`. It is only a valid value to the `realresolution` parameter of the discipline. An error will be issued if `none` is passed as an argument to `-wreal_resolution`.

- **Defining Global wreal Resolution Function Using the AMSCB connectmap Card.**
For a `connectmap` card, if the `discipline` parameter is absent, the resolution function specified by the `realresolve` parameter applies globally to all disciplines that are not covered by other `connectmap` cards.

Consider the following example:

```
amsd{  
  
    connectmap realresolve=max  
  
    connectmap discipline="disc1 disc2" realresolve=min  
  
    connectmap discipline="disc3 disc4" realresolve=sum  
  
}
```

In the above example:

- The `disc1` and `disc2` disciplines will have the resolution function `min`.
- The `disc3` and `disc4` disciplines will have the resolution function `sum`.
- Any other discipline will have the resolution function `max`.

- The resolution function defined through the `-wreal_resolution` command-line option has precedence over the one defined through the AMSCB `connectmap` card. So, if both methods are used and they specify different resolution functions, the global `wreal` resolution function takes its value from the command-line option and a warning is printed on the screen.
- If no global `wreal` resolution function is specified using these two methods, it is set to the `default` `wreal` resolution function.

Order of Precedence for Determining the Resolution Function on a Net

If a discipline has a resolution function defined in the discipline definition and also has a resolution function defined in the AMS Control Block using the `connectmap` card, the `connectmap` card resolution function will take precedence and will be used as the resolution function for all nets of that discipline. This allows you to override the resolution function without changing the discipline definition.

Following is the order of precedence for determining the resolution function on a net:

1. Explicitly declared resolution on a net it is connected to (typed `wreal` or SV-RNM nettype)
2. Resolution function explicitly defined in the AMS Control Block using the `connectmap` card
3. Resolution function explicitly defined in the nets discipline definition
4. Resolution function of the discipline of the net to which a given net is connected
5. Global resolution function specified with the `-wreal_resolution` command-line option
6. Global resolution function specified with the `connectmap` card (without the `discipline` parameter)
7. The `default` global resolution function

Rules to Determine the Resolved Resolution Function for Connected `wreal` Nets

During elaboration, the connected `wreal` nets are collapsed into a single simulated `wreal` net with a single resolution function. The resolved value of this simulated `wreal` net is determined by considering all its drivers and processing their values through the resolved resolution function.

The set of rules listed below are followed to determine the resolved resolution function.

- If all connected `wreal` nets have the same resolution function, the resolved resolution function is that resolution function.

The connected `wreal` nets need not have the same discipline as long as the resolution function of each net is the same.

- If two or more `wreal` nets are connected together and they do not have the same `wreal` resolution function, an error occurs.
- If a `wreal` net does not belong to any discipline or its discipline does not have a specified resolution function, the resolution function of the `wreal` net is the global resolution function. When a `wreal` net with the global resolution function is connected to other `wreal` nets, its resolution function is ignored for the purpose of determining the resolved resolution function for the connection.
- If none of the `wreal` nets in a connection has a specified resolution function, the resolved resolution function for the connection will be the global resolution function.

The table below summarizes how the resolved resolution function is determined from the resolution functions of the three `wreal` nets: `w1`, `w2`, and `w3`.

w1	w2	w3	Resolved
avg	avg	avg	avg
avg	UNSPECIFIED	UNSPECIFIED	avg
avg	UNSPECIFIED	sum	ERROR
UNSPECIFIED	UNSPECIFIED	UNSPECIFIED	GLOBAL

Related Topic

- [Predefined `wreal` Resolution Functions](#)

Predefined `wreal` Resolution Functions

The following predefined `wreal` resolution functions can be used to [define](#) `wreal` resolution functions for a discipline:

- **default:** The `default` resolution function algorithm for `wreal` nets that have more than one driver is as follows:

Conditions	Resolution
All drivers are driving <code>wrealZState</code>	Drive the receivers using <code>wrealZState</code>
Exactly one driver is not driving <code>wrealZState</code>	Drive the receivers using the only non- <code>wrealZState</code> value
More than one driver is not driving <code>wrealZState</code>	Drive the receivers using <code>wrealXState</code> and issue a run-time warning message
Any driver is driving <code>wrealXState</code>	Drive the receivers using <code>wrealXState</code>

Here is an example of how the elaborator determines the resolved value of two drivers, `wreal1` and `wreal2` using the `default` resolution function.

<code>wreal1</code> Value	<code>wreal2</code> Value	Resolved Value
<code>wrealZState</code>	<code>wrealZState</code>	<code>wrealZState</code>
1.23	<code>wrealZState</code>	1.23
<code>wrealZState</code>	3.33	3.33
1.23	1.23	<code>wrealXState</code> ¹
1.23	3.33	<code>wrealXState</code> ¹
1.23	<code>wrealXState</code>	<code>wrealXState</code>
<code>wrealXState</code>	3.33	<code>wrealXState</code>
<code>wrealXState</code>	<code>wrealZState</code>	<code>wrealXState</code>

¹ If there is more than one non-`wrealZState` driver, the software issues a run-time warning message.

- **fourstate:** The `fourstate` resolution function algorithm for `wreal` nets that have more than

one driver is the same as the Verilog 4-state logic resolution algorithm. It is similar to the `default` algorithm except that identical drivers resolve to the driven value instead of to `wrealXState`.

The `fourstate` resolution function algorithm is as follows:

Conditions	Resolution
All drivers are driving <code>wrealZState</code>	Drive the receivers using <code>wrealZState</code>
Exactly one driver is not driving <code>wrealZState</code>	Drive the receivers using the only non- <code>wrealZState</code> value
More than one driver is not driving <code>wrealZState</code>	- When all non- <code>wrealZState</code> drivers drive the same value, drive the receivers using that value and issue a run-time warning message - When all non- <code>wrealZState</code> driver do not drive the same value, drive the receivers using <code>wrealXState</code> and issue a run-time warning message
Any driver is driving <code>wrealXState</code>	Drive the receivers using <code>wrealXState</code>

Here is an example of how the elaborator determines the resolved value of two drivers, `wreal1` and `wreal2` using the `fourstate` resolution function.

wreal1 Value	wreal2 Value	Resolved Value
<code>wrealZState</code>	<code>wrealZState</code>	<code>wrealZState</code>
1.23	<code>wrealZState</code>	1.23
<code>wrealZState</code>	3.33	3.33
1.23	1.23	1.23
1.23	3.33	<code>wrealXState</code> ¹
1.23	<code>wrealXState</code>	<code>wrealXState</code>

wrealXState	3.33	wrealXState
wrealXState	wrealZState	wrealXState

1 If there is more than one non-wrealZState driver, the software issues a run-time warning message.

- **sum:** The `sum` resolution function algorithm for **wreal** nets that have more than one driver resolves to the summation of all the drivers, ignoring `wrealZState` drivers. If any drivers are driving `wrealXState`, the elaborator resolves the value to `wrealXState`. Any nets driving `wrealZState` do not contribute to the resolved value.

The `sum` resolution function algorithm for **wreal** nets that have more than one driver is as follows:

Conditions	Resolution
All drivers are driving wrealZState	Drive the receivers using <code>wrealZState</code>
Any driver is driving <code>wrealZState</code>	Drive the receivers with the summation of all non- <code>wrealZState</code> values (that is, ignore all <code>wrealZState</code> drivers)
No driver is driving <code>wrealZState</code>	Drive the receivers with the summation of all the drivers
Any driver is driving wrealXState	Drive the receivers using <code>wrealXState</code>

Here is an example of how the elaborator determines the resolved value of two drivers, `wreal1` and `wreal2` using the `sum` resolution function.

wreal1 Value	wreal2 Value	Resolved Value
<code>wrealZState</code>	<code>wrealZState</code>	<code>wrealZState</code>
1.23	<code>wrealZState</code>	1.23
<code>wrealZState</code>	3.33	3.33
1.23	1.23	2.46
1.23	3.33	4.56

1.23	wrealXState	wrealXState
wrealXState	3.33	wrealXState
wrealXState	wrealZState	wrealXState

- **avg:** The `avg` resolution function algorithm for `wreal` nets that have more than one driver is as follows:

Conditions	Resolution
All drivers are driving <code>wrealZState</code>	Drive the receivers using <code>wrealZState</code>
Any driver is driving <code>wrealZState</code>	Drive the receivers with the average of all non- <code>wrealZState</code> values (that is, ignore all <code>wrealZState</code> drivers)
No driver is driving <code>wrealZState</code>	Drive the receivers with the average of all the drivers
Any driver is driving <code>wrealXState</code>	Drive the receivers using <code>wrealXState</code>

Here is an example of how the elaborator determines the resolved value of two drivers, `wreal1` and `wreal2` using the `avg` resolution function.

wreal1 Value	wreal2 Value	Resolved Value
<code>wrealZState</code>	<code>wrealZState</code>	<code>wrealZState</code>
1.23	<code>wrealZState</code>	1.23
<code>wrealZState</code>	3.33	3.33
1.23	1.23	1.23
1.23	3.33	2.28
1.23	<code>wrealXState</code>	<code>wrealXState</code>
<code>wrealXState</code>	3.33	<code>wrealXState</code>
<code>wrealXState</code>	<code>wrealZState</code>	<code>wrealXState</code>

- **min:** The `min` resolution function algorithm for `wreal` nets that have more than one driver is as

follows:

Conditions	Resolution
All drivers are driving wrealZState	Drive the receivers using <code>wrealZState</code>
Any driver is driving <code>wrealZState</code>	Drive the receivers with the minimum of all non- <code>wrealZState</code> values (that is, ignore all <code>wrealZState</code> drivers)
No driver is driving <code>wrealZState</code>	Drive the receivers with the minimum of all the drivers
Any driver is driving wrealXState	Drive the receivers using <code>wrealXState</code>

Here is an example of how the elaborator determines the resolved value of two drivers, `wreal1` and `wreal2` using the `min` resolution function.

wreal1 Value	wreal2 Value	Resolved Value
<code>wrealZState</code>	<code>wrealZState</code>	<code>wrealZState</code>
1.23	<code>wrealZState</code>	1.23
<code>wrealZState</code>	3.33	3.33
1.23	1.23	1.23
1.23	3.33	1.23
1.23	<code>wrealXState</code>	<code>wrealXState</code>
<code>wrealXState</code>	3.33	<code>wrealXState</code>
<code>wrealXState</code>	<code>wrealZState</code>	<code>wrealXState</code>

- **max:** The `max` resolution function algorithm for **wreal** nets that have more than one driver is as follows:

Conditions	Resolution
All drivers are driving wrealZState	Drive the receivers using <code>wrealZState</code>

Any driver is driving <code>wrealZState</code>	Drive the receivers with the maximum of all non- <code>wrealZState</code> values (that is, ignore all <code>wrealZState</code> drivers)
No driver is driving <code>wrealZState</code>	Drive the receivers with the maximum of all the drivers
Any driver is driving <code>wrealXState</code>	Drive the receivers using <code>wrealXState</code>

Here is an example of how the elaborator determines the resolved value of two drivers, `wreal1` and `wreal2` using the `max` resolution function.

wreal1 Value	wreal2 Value	Resolved Value
wrealZState	wrealZState	wrealZState
1.23	wrealZState	1.23
wrealZState	3.33	3.33
1.23	1.23	1.23
1.23	3.33	3.33
1.23	wrealXState	wrealXState
wrealXState	3.33	wrealXState
wrealXState	wrealZState	wrealXState

Related Topics

- [Selecting a wreal Resolution Function](#)
- ["Real Nets with More than One Driver"](#)

Verilog-AMS–Based RNM for Wreals

The Verilog-AMS based real number modeling (RNM) solution facilitates high performance and reasonably accurate modeling of analog behavior to aid verification of mixed-signal designs. You can use the Verilog-AMS based RNM functionality to write:

- SystemVerilog-compliant RNMs using disciplineless wreal nettypes (discrete time domain)
- Portable Verilog-AMS wreal models for use in SystemVerilog (SV)

This capability enables you to reuse your code according to the changes to the SV modeling features.

Related Topics

- [Writing RNMs Using Disciplineless Wreal Nets](#)
- [Writing Portable Verilog-AMS Wreal Models for Use in SV](#)
- [SystemVerilog Real Number Modeling](#)

Writing RNMs Using Disciplineless Wreal Nets

The Verilog-AMS based real number modeling (RNM) solution facilitates high performance and reasonably accurate modeling of analog behavior to aid verification of mixed-signal designs. You can use disciplineless wreal nets to resolve multiple drivers on wreal signals to suit the expected changes in SV.

To accomplish this, use the six new disciplineless wreal net types--wrealavg, wrealsum, wrealmax, wrealmin, wrealldriver, and wreal4state--added to Verilog-AMS. These disciplineless wreal nettypes enable you to write wreal models that align with the SV-RNM requirement.

Example

To declare disciplineless wreal nets that have resolution associated with them, write:

```
wrealsum wr_sum_1;  
wrealavg wr_avg_1;  
wrealmin wr_min_1;
```

Here is a full code example using the new wreal nettypes:

```
module top();  
  
    //new discipline-less with resolution function avg  
    wrealavg real_wire;  
    source1 I12 (real_wire);  
    source2 I22 (real_wire);  
    sink I32(real_wire);  
  
endmodule
```

Related Topics

- [Verilog-AMS–Based RNM for Wreals](#)

Writing Portable Verilog-AMS Wreal Models for Use in SV

The disciplineless net types combine resolution functions with data type declaration. This new use model enables you to write Verilog-AMS models that can be fully transported to SV.

To write a portable Verilog-AMS wreal model:

1. Create wreal models in Verilog-AMS using the new disciplineless nettypes.
2. Rename your model file extension from `.vams` to `.sv`.
3. Add a library import (provided by Cadence) to your SV model in Verilog-AMS. This library brings in the SV nettypes that correspond to the current disciplineless wreal nettypes.

The Cadence library uses built-in resolution functions for Cadence simulators. For other simulators, you will also need a library of resolution functions.

4. Add the Cadence SV library to your simulation command.

To build portable models, your Verilog-AMS models should:

- Not contain analog procedural blocks, discipline definition or usage, and electrical nets.
- Only use the new nettypes, for example `wrealsum`, for multiple driver resolution function.

For example:

```
module cc_amplifier (i_in, vout); // Current-controlled amplifier

    input i_in; wrealsum i_in; // Input using discipline-less wreal
    output vout; wreal vout; // Output using built-in type
    parameter real gain = 1; // voltage gain constant
    assign vout = gain * i_in; // update voltage output

endmodule
```

Related Topics

- [Verilog-AMS–Based RNM for Wreals](#)
- [Writing RNMs Using Disciplineless Wreal Nets](#)

Wreal Concatenation Expressions

The AMS Designer simulator supports the following wreal concatenation expressions:

- Concatenation expressions involving wreals in port maps
- Connections of concatenation expressions, which contain wreal nets to wreal or logic wire ports using R2L connect modules
- Selective coercion of nets used in concatenation expressions that contain nets to wreal

The following use cases are not supported:

- The use of concatenation expressions involving wreals to assign to a wreal array is not supported.

For example, use the following:

```
wreal d[0:1];  
wreal a, b;  
assign d[0] = a; //legal assignment  
assign d[1] = b; //legal assignment
```

instead of the following:

```
wreal d[0:1];  
wreal a, b;  
assign d = {a, b}; //illegal assignment
```

- Explicit declaration of wreal in non-collapsible concatenation is not allowed and results in an error.

A port connection is collapsible if the upper and lower connections are nets. For connections of selects of packed or unpacked net arrays, the selects must have constant indices to be collapsible. Amongst other things, the presence of variables or constant expressions on either side of the connections makes the port connections non-collapsible.

A non-collapsible concatenation expression contains one or more non-collapsible nets including reg, expression, parameter, and constant objects. The non-collapsible nets are not coerced and coercion does not proceed through such concatenated expressions. R2L IEs are inserted between these expressions/nets and wreal ports.

Example of a non-collapsible concatenation expression:

```
wire w1, w2;  
reg val1, val2;  
parameter integ g, h;
```

```
{1'b1, 1'b0}, //constants  
{val1, val2}, //regs  
{g, h}, //parameters  
{cos(1), 1+1}, //expressions  
{w1, wrealZState}, //constants  
{w1, w2, val1}, //regs
```

- Concatenation expressions that contain wreal nets and non-collapsible expressions, such as constants and variables, are not supported.
- Connection of concatenation expressions that contain wreal nets to non-collapsible expressions, such as [SV](#) variable ports, Verilog reg ports, and constants, are not supported.
- The following nets used in concatenation expressions are not coerced to wreal:
 - Nets containing non-collapsible elements
 - If a concatenation expression containing nets is the upper or lower expression of a port, and if the other expression is non-collapsible

Related Topics

- [Wreal Connections with Concatenated Nettypes to Array of Instances](#)
- [Wreal Connections with Concatenated Wires and Nettype Ports](#)

L2R and R2L Connect Modules

Logic-to-Real connect modules (L2R) are similar to Logic-to-Electrical (L2E) connect modules. They are also declared with the keyword `connectmodule` and contain two ports: one a scalar wreal net (R) and the other a scalar logic net (L). They must be flat and can contain no sub-hierarchies. They take their parameter overrides from the connect rules, just like the L2E connect modules. The internals of the L2R connect modules convert the driven value of one side of the logic/real boundary to the appropriate data type, and drive it to the other side of the boundary.

The L2R and R2L connect modules are defined for unidirectional ports (input and output) and only pass driver information in a single direction as indicated by the port directions.

Here is an example of an L2R connect module:

```
connectmodule L2R(L, R);
  input L; \logic L;
  output R; wreal_dsp R;

  parameter real vsup = 1.8 from (0:inf);
  parameter real vlo = 0;
  parameter real vhi = vsup from (vlo:vsup);
  parameter real vtlo = vsup / 3;
  parameter real vthi = vsup /1.5;

  wire [31:0] L_val;
  reg [1:0] L_code;
  real L_real;

  initial
    begin
      $BIE_input_strength(L, L_val);
    end

  // Determine the value and strength of L and convert to a real number
  always
    begin
      L_code = L_val & 2'b11;

      case (L_code)
        2'b00: L_real = vlo;
        2'b01: L_real = vsup;
        2'b11: L_real = `wrealXState
        2'b10: L_real = `wrealZState;
      endcase

      @(L_val)
    end

  // drive the converted value back onto the R output pin

assign R = L_real;

endmodule64ce7
```


Here is an example of an R2L connect module:

```
connectmodule R2L(L, R);
  output L; \logic L;
  input R; wreal_dsp R;

  parameter real vsup = 1.8 from (0:inf);
  parameter real vlo = 0;
  parameter real vhi = vsup from (vlo:vsup);
  parameter real vtlo = vsup / 3;
  parameter real vthi = vsup /1.5;

  wreal R_val;
  reg R_logic;

  initial
    begin
      $BIE_input_real(R, R_val);
    end

  // Determine the value of R and convert to a logic value
  always
    begin
      if(R_val >= vthi)
        R_logic = 1'b1;
      else if (R_val <= vtlo)
        R_logic = 1'b0;
      else if(R_val === `wrealZState)
        R_logic = 1'bz;
      else
        R_logic = 1'bx;

      @ (R_val);
    end

  // drive the converted value back onto the output L pin
  assign L = R_logic;

endmodule
```

Currently, R2L/L2R connect modules with mixed-signal and/or mixed-language interactions are not supported.

Inherited connections are allowed inside the L2R and R2L connect modules with the following limitations:

- The net you are inheriting from must be a wreal net.
- The discipline of the wreal net you are inheriting from must be digital. [E2R](#) connect modules are not inserted into L2R connect modules.

- The connect module cannot drive a value onto the inherited connection.

If any of these conditions is violated, an error will occur.

The discipline restriction also affects the logic nets. Logic nets in the connect modules cannot inherit from nets with incompatible discipline.

Related Topics

- [Port Connections through Real-to-Logic Connect Modules](#)
- [Inherited Connections in R2L/L2R Connect Modules](#)
- [Aliasing Nets to Hierarchical Nets in L2R/R2L Interface Elements](#)
- [L2R/R2L Insertion in Multiple-Supply Designs](#)

Port Connections through Real-to-Logic Connect Modules

The following types of port connections can be made through Real-to-Logic (R2L) connect modules:

- A wreal upper connection to SystemVerilog ([SV](#)) logic output ports
- An SV logic variable or expression upper connection to wreal input ports
- SV real variables or expressions to Verilog-compatible logic nets

Note that this feature does not support the following scenarios:

- A wreal upper connection to an SV logic input port
- An SV logic variable upper connection to a wreal output port

You can insert R2L connect modules in SV scopes, when the upper or lower connection is a variable or expression, and the other is a net. For example, you can connect an [SV](#) real variable to a Verilog net input port with logic data type by inserting an R2L connect module. This converts the value of the real variables to a Verilog 4-state logic value, and drives that value onto the net. When you insert an R2L connect module, the mixed-signal elaborator connects the ports.

This feature supports the following port connection scenarios:

The term "variable" refers to a variable identifier such as a scalar, whole array, constant bit select, and constant part select. The term "expression" represents other possible expressions including constants, concatenations, and mutable selects.

- An SV scalar real variable or expression upper connection connected to an SV or Verilog-AMS (VAMS) scalar logic net lower connection port
 - If the lower connection port is:
input: Inserts R2L connect module
output: Inserts L2R connect module
 - If the upper connection is not an lvalue expression (can be on the left-hand side of a continuous assignment), it is an error
inout: Displays an error
- An SV scalar logic variable or expression upper connection connected to a VAMS scalar wreal net lower connection port
 - If the lower connection port is:
input:insert L2R connect module
output:error
inout:error
- A VAMS scalar wreal net upper connection connected to an SV scalar logic variable lower connection port
 - If the lower connection port is:
input: error
output: insert L2R connect module
inout: error
- An SV or VAMS scalar logic net upper connection connected to an SV scalar real variable lower connection port
 - If the lower connection port is:
input: insert L2R connect module
output: insert R2L connect module
inout:error
- An SV unpacked real variable array or real array expression upper connection connected to an SV or VAMS packed vector logic net lower connection port
 - If the lower connection port is:
input:insert R2L connect modules for each element (the lower and upper connection expressions must be of the same size)
output:insert L2R connect modules for each element (the lower and upper connection expressions must be of the same size)

- If the upper connection is not an lvalue expression (can be on the left-hand side of a continuous assignment), it is an error
inout:error
- An [SV](#) packed logic variable array or packed logic expression upper connection connected to a VAMS unpacked wreal array net lower connection port
 - If the lower connection port is:
input:insert R2L connect modules for each element (the lower and upper connection expressions must be of the same size)
output:error
inout:error
- A VAMS unpacked wreal array net upper connection connected to an SV packed logic variable array lower connection port
 - If the lower connection port is:
input:error
output:insert L2R connect modules for each element (the lower and upper connection expressions must be of the same size)
inout:error
- An SV or VAMS packed logic array net upper connection connected to an SV unpacked real variable array lower connection port
 - If the lower connection port is:
input:insert L2R connect modules for each element (the lower and upper connection expressions must be of the same size)
output:insert R2L connect modules for each element (the lower and upper connection expressions must be of the same size)
inout:error

Other scenarios involving real connections to logic that are not supported in the SV LRM explicitly, results in an error. Cases where an R2L is required to drive a logic variable also result in an error.

Related Topics

- [L2R and R2L Connect Modules](#)
- [L2R/R2L Insertion in Multiple-Supply Designs](#)

Inherited Connections in R2L/L2R Connect Modules

You can create Real-to-Logic (R2L)/Logic-to-Real (L2R) connect modules that use inherited connections to read global power and ground values, so that you can use a dynamic supply to control the wreal/logic conversion threshold. Similarly, appropriate connect rules are created along with the new connect modules.

Here is an example of an L2R connect module that uses inherited connection:

```

`include "disciplines.vams"
`timescale 1ns / 1ps

connectmodule L2R_inhconn(Rout, Lin);
input Lin;
output Rout; wreal Rout;

// Inherited vdd! and vss!
wreal
(* integer inh_conn_prop_name="vdd";
integer inh_conn_def_value="cds_globals.\\vdd! "; *) \vdd! ;

wreal
(* integer inh_conn_prop_name="vss";
integer inh_conn_def_value="cds_globals.\\vss! "; *) \vss! ;
parameter real vsup_min=0.5 from (0:inf); // min supply for normal operation
parameter real vlo = 0; // logic low voltage
reg supOK;
real L_conv;

initial begin
L_conv = `wrealZState;
end

always begin
if ( \vdd! - \vss! > vsup_min) supOK = 1'b1;
else supOK = 1'b0;
@(\vdd! , \vss! );
end

// Determine the value of L and convert to a real value
always begin
if ( supOK ) begin
case (Lin)
1'b0:
L_conv = \vss! ;
1'b1:
L_conv = \vdd! ;
1'bz:
L_conv = `wrealZState;
default:
L_conv = `wrealXState;
endcase // case(L_code)
end
else
L_conv = `wrealXState;
@(Lin, supOK);
end

// drive the converted value back onto the output R pin
assign Rout = L_conv;
endmodule

```

Here is an example R2L connect module that uses inherited connection:

```
`include "disciplines.vams"
`timescale 1ns / 1ps

connectmodule R2L_inhconn(Rin, Lout);
output Lout;
input Rin; wreal Rin;

// Inherited vdd! and vss!
wreal
(* integer inh_conn_prop_name="vdd";
integer inh_conn_def_value="cds_globals.\\vdd! "; *) \vdd! ;

wreal
(* integer inh_conn_prop_name="vss";
integer inh_conn_def_value="cds_globals.\\vss! "; *) \vss! ;
parameter real vsup_min=0.5 from (0:inf); // min supply for normal operation
parameter real vthi=1/1.5 from (0:1); // frac. for high thres (def=2/3)
parameter real vtlo=vthi/2 from (0:vthi); // frac. for low thres (def=1/3)
parameter real txdel=0.8n from (0:1m); // time midrange til output X
real vsup, vtl, vth, txdig=txdel/1n;
reg supOK, Xin=0, R_conv = 1'bz;

always begin
vsup = \vdd! - \vss! ;
if ( vsup > vsup_min)
begin
supOK <= 1'b1 ;
vtl <= vsup * vtlo + \vss! ;
vth <= vsup * vthi + \vss! ;
end
else begin
supOK <= 1'b0 ;
end
@(\vdd! , \vss! );
end

// Determine the value of R and convert to a logic value
always begin
if ( supOK ) begin
if(Rin >= vth)
begin R_conv = 1'b1; Xin = 0; disable GoToX; end
else if (Rin <= vtl)
begin R_conv = 1'b0; Xin = 0; disable GoToX; end
else if(Rin == `wrealZState)
begin R_conv = 1'bz; Xin = 0; disable GoToX; end
else
Xin = 1;
end
else
Xin = 1;
```

```
@(Rin, supOK, vth, vtl);  
end  
  
// see if it is a stable X  
always @ (posedge(Xin)) begin: GoToX  
#(txdig)  
if (Xin == 1 ) R_conv = 1'bx;  
end  
  
// drive the converted value back onto the output L pin  
assign Iout = R_conv;  
endmodule
```

For L2R/R2L connect modules that use inherited connection:

- The two global nets `cds_globals.vdd!` and `cds_globals.vss!` must be wreal.
- It is an error if the two global nets are not wreal and need IE insertion again.

Related Topics

- [L2R and R2L Connect Modules](#)
- [Port Connections through Real-to-Logic Connect Modules](#)

Aliasing Nets to Hierarchical Nets in L2R/R2L Interface Elements

You can alias a local net to a hierarchical net within Logic-to-Real (L2R) and Real-to-Logic (R2L) interface elements. To do this, use the `$real_net_alias` system function:

```
real_net_alias_system_function ::=  
  
    $real_net_alias (real_net_reference, hierarchical_reference_string)
```

Here, `real_net_reference` is a local net that is declared in the module containing the system function call. The local net can be a scalar or a vector wreal net, a wreal array or SV-UDN real net array. If `real_net_reference` is a vector, it references the full vector. Part-select or bit-select of array is not supported.

`hierarchical_reference_string` can be a string literal or a string parameter containing a hierarchical reference to the wreal net, a scalar element of a vector wreal net, a wreal array, or SV-UDN real net array. The array reference can be bit-select or part-select. In bit select array reference, the index must be an integer constant value.

In SV-UDN real net array connections, both the `real_net_reference` and `hierarchical_reference_string` arguments must be of the same nettype. And, the `$real_net_alias` system function does not support SV-UDN nets with struct type; and, returns zero

value in this case.

When both the `real_net_reference` and `hierarchical_reference_string` arguments are arrays, each bit of the first argument connects to each of the second argument. When `real_net_reference` is array and `hierarchical_reference_string` is scalar or bit-select, each bit of the array would connect to the scalar or bit-select. And, when the `hierarchical_reference_string` is array, the width of both arguments should be identical.

The `$real_net_alias` system function returns 1 if `hierarchical_reference_string` points to a valid discrete wreal net and returns 0 (zero) otherwise.

The `$real_net_alias` system function can only be used in a digital block. You can use the `$real_net_alias` system function in both connect modules and Verilog-AMS modules.

You can use the Verilog-AMS function, `$analog_node_alias`, to alias a node to a hierarchical node within connect modules and regular modules.

In the following example both `real_net_reference` and `hierarchical_reference_string` arguments are wreal arrays, and the `real_net_reference` array reference, `top.top_sig[3:1]` is part-select.

```
// top.sv
module top;
    wreal top_sig[3:0];

    assign top_sig[0] = 0.1;
    assign top_sig[1] = 1.2;
    assign top_sig[2] = 2.3;
    assign top_sig[3] = 3.4;

    foo f1 ();
endmodule

module foo;
    wreal foo_sig[2:4];

    reg r;
    initial begin
        r = $real_net_alias(foo_sig, "top.top_sig[3:1]"); // support part selection
        #3 $display(r,,foo_sig[2],, foo_sig[3],, foo_sig[4]);
    end
endmodule
```

Limitations

- Real net defined in a VHDL module cannot be specified as target for `hierarchical_reference_string`
- The `hierarchical_reference_string` cannot have a VHDL top OOMR. For instance, in the following example, `top` cannot be a VHDL module

```
hierarchical_reference_string = "top.child.leaf.net"
```
- Verilog upward OOMR is not supported

For more information on the `$real_net_alias` and `$analog_node_alias` functions, see [Node/Net Aliasing System Functions](#).

Interaction between L2R/R2L and L2E/E2L Interface Elements on a Hierarchical Net

When a hierarchical net has segments where one or more of those segments are digital-logic type, digital-real type, and analog-real type, it requires an R-L-E interaction. The following interactions are possible between different segments of such a net:

- Logic-to-Electrical (L2E)
- Wreal-to-Electrical (R2E)
- Wreal-to-Logic (R2L)

The R-L-E interaction is enabled by default. If a hierarchical signal has both R2L/L2R and L2E/E2L interface elements (IEs), then driver-receiver segregation (DRS) is not performed on any of those IEs. This means that all AMS IEs inserted across that signal use the SIE technology (see [Using the Strength-Based Interface Element \(SIE\)](#)).

R-L-E optimization is supported in the AMS-CPF flow. If an isolation device is required on the L-R or L-E boundary, it is applied on the logic side of the connect module. However, if an isolation device is required on the E-R boundary, it is ignored and a warning message is generated.

Related Topics

- [L2R and R2L Connect Modules](#)
- [Electrical Supply in Real-to-Logic and Logic-to-Real Interface Elements](#)

L2R/R2L Insertion in Multiple-Supply Designs

To use the in-built Real-to-Logic (R2L) or Logic-to-Real (L2R) connect modules for multiple-supply designs, you must specify the `xrun` option `-R2L_MSUP`. For instance, in designs where a logic net with discrete discipline, say `discipline_1`, connects to a wreal net with discrete discipline, say `discipline_2`, the elaborator treats these disciplines as not compatible and inserts an R2L/L2R connect module.

For example:

```
module dig_in ( in, out );
input in;
output out;
assign #10 out = in;
endmodule

module top;
wreal wr_drv;
dig_in dsink ( wr_drv, wo );
endmodule

amsd{
ie vsup=5 discipline=discipline_1
ie vsup=1.8 cell=dig_in discipline=discipline_2
}
```

The R2L/L2R connect module is inserted based solely on the discrete discipline of the scalar logic net, when it is available. However, if a scalar logic net does not have a discrete discipline and scalar wreal net has, then an R2L/L2R connect module is inserted based on the discrete discipline of the scalar wreal net.

Related Topics

- [L2R and R2L Connect Modules](#)

Electrical Supply in Real-to-Logic and Logic-to-Real Interface Elements

The following built-in inherited connect modules and supply sensitivity modules that accept electrical supply have been added to Xcelium:

- `R2L_electrical_inh`
- `L2R_electrical_inh`
- `RL_bidir_electrical_inh`

- [R2L_electrical_ss](#)
- [L2R_electrical_ss](#)
- [RL_bidir_electrical_ss](#)

These connect modules can be located at `<your_install_directory>/tools.
<plat>/affirma_ams/etc/connect_lib/connectLib`.

To use these connect modules, you must specify the xrun option `-r2l_electrical_inh`. When this option is specified, the simulator uses the R2L IEs with electrical inherited connections instead of R2L IEs with wreal inherited connections.

For built-in inherited connect modules and supply sensitivity modules to work, the design should use the same kind of supply, that is, electrical or wreal.

Related Topics

- [Aliasing Nets to Hierarchical Nets in L2R/R2L Interface Elements](#)
- [L2R and R2L Connect Modules](#)
- [Inherited Connections in R2L/L2R Connect Modules](#)

Virtuoso Visualization and Analysis in xrun and ADE Flows for Simulations with Real Number Models

You can use the Cadence® Virtuoso® Visualization and Analysis tool with AMS Designer for wreal simulations involving the digital solver, through the Xcelium *xrun* and the Virtuoso ADE flows.

Virtuoso Visualization and Analysis XL supports the plotting of WREAL (wire-real) signals, where the WREAL signals are by default plotted in the sample and hold plot type.

Related Topic

- [Plotting Wreal Signals](#)

Real Number Modeling Debug Options

You can use the `xrun` command with the following options to debug issues in designs using real number modeling:

- `+wreal_res_info`: Prints a report of all `wreal` nets in the design and their resolution functions.
- `-rnm_info`: Prints a report of all `.vams`, `wreal`, `SV-wreal` nettypes, and `SV-UDN` structs within the design, including nettype declaration and coercion information of these nets.
- `-show_rnm_cover`: Prints the file name and line number of RNM element inside a coverage group that enabled the DMSO license.

Additionally, you can also use the `-show_rnm_rand` option to get the filename and line number of real randomization that enabled the DMSO license.

Related Topics

- [Real Number Modeling](#)
- [Using the `wreal` Data Type](#)

wreal to VHDL Connections

In mixed-signal designs, to map VHDL analog X/Z states with the Verilog `wrealZ/X` states use the `xrun` command-line option, `-wreal2vhdlmap file`.

You can specify the VHDL/`wreal` X and Z state mappings in a `file`. The template format is in the form of `wreal2vhdlmap card` which defines the mapping for the specified VHDL data type. Multiple `wreal2vhdlmap` cards can be specified in the file. The format of the `wreal2vhdlmap` card is shown below:

```
wreal2vhdlmap type=[ "default" | "<vhdl type>" ] \  
  
wrealXState = "<vhdl primary xstate> <vhdl secondary xstate> ....." \  
  
wrealZState = "<vhdl primary zstate> <vhdl secondary zstate>,...."
```

When `type` is set to `default`, it indicates the default mapping, otherwise, `type` specifies the VHDL type or subtype that needs to be mapped. If a type-based match is not found, the default mapping is used. You can specify only one `wreal2vhdlmap` card with `type=default`.

Currently, a simulation run supports only five `wreal2vhdlmap` card entries.

The `wrealXState` and `wrealZState` arguments support a list of values to be mapped. The first element in the list is considered the primary value map and the remaining elements, if specified, are considered as secondary value maps. The primary value map is significant because it defines the value mapping for both Verilog driving VHDL and VHDL driving Verilog. The secondary maps are only used for mapping VHDL driving Verilog. In this form, a one-to-one mapping is supported for Verilog driving VHDL, whereas a many-to-one mapping is supported for VHDL driving Verilog. The `wrealXState` parameter defines the X state maps while the `wrealZState` parameter defines the Z state mapping.

When VHDL is driving a primary or secondary value, the value is mapped to the `wrealXState` or `wrealZState` value in the Verilog domain. Note that the primary and secondary values specified in `wrealXState` and `wrealZState` parameter must be unique, otherwise, a fatal error is reported.

The `type` argument (except `type=default`) is used when an exact match is desired. `wreal2vhdlmap` card also supports derived mapping. Derived mapping can be performed by specifying the `derived_type` argument in the `wreal2vhdlmap` card, as shown below.

```
wreal2vhdlmap derived_type="<vhdl type>"\
wrealXState = "<vhdl xstate1> <vhdl xstate2> ....." \
wrealZState = "<vhdl zstate1> <vhdl zstate2> ,...."
```

In this mode, the search is extended to the type decomposition of the actual. A match occurs when the specified `derived_type` matches one of the decomposed types of the actual. Note that the mapping algorithms find the best map where the order of precedence is `type` mappings, `derived_type` mappings and `default` mappings. For `derived_type` mappings the lookup algorithm binds to the `derived_type` mapping closest with respect to the decomposition to the actual type.

Example:

The following example shows a VHDL package with type declaration to be mapped to `wreal`. The constant values represent the VHDL values to be mapped to `wreal x` and `z` states. This package `myPkg` is compiled into the library `myLib`.

```
package is myPkg
type analog_ut is range real'low to real'high;
type analog_vector_ut is array (natural range <>) of analog_ut;
constant ANALOG_U : analog_ut := analog_ut'left;
constant ANALOG_X : analog_ut := analog_ut(real(analog_ut'left) / 2.0);
constant ANALOG_Z : analog_ut := analog_ut(real(analog_ut'left) / 4.0);
end myPkg;
```

Following is the file that contains the `wreal2vhdlmap` template for the VHDL type above:

```
wreal2vhdlmap type="myLib.myPkg.analog_ut" wrealXState = "myLib.myPkg.ANALOG_X  
myLib.myPkg.ANALOG_U" \  
  
wrealZState = "myLib.myPkg.ANALOG_Z"
```

This template states that for mixed language boundary net connections of `myLib.myPkg.analog_ut` connecting to Verilog real, the following mapping occurs:

- Verilog driving a `wrealXState` value will be mapped to a `myLib.myPkg.ANALOG_X` VHDL value (primary value)
- Verilog driving a `wrealZState` value will be mapped to a `myLib.myPkg.ANALOG_Z` VHDL value.
- VHDL driving `myLib.myPkg.ANALOG_X` or `myLib.myPkg.ANALOG_U` value will be mapped to a `wrealXState` Verilog value.

VHDL driving a `myLib.myPkg.ANALOG_Z` value will be mapped to a `wrealZState` Verilog value.

Related Topic

- [wreal Nets at Mixed-Language Boundaries](#)

XRUN Invocation for Mixed-Signal Designs

The *xrun* utility lets you run the Xcelium™ Simulator using either the single-core or multi-core engine. This utility accepts files from different simulation languages, such as Verilog, Verilog-AMS, SystemVerilog (SV), VHDL, VHDL-AMS, SPICE, and Spectre, as well as mixed-language designs.

The `xrun` command supports both single-step and multi-step invocation modes for compiling, elaborating, and simulating your design:

- **Single-step xrun invocation:** You can specify all input files and all command line options in a single command line that compiles the design using appropriate compilers and then the tool automatically invokes elaboration and simulation.
- **Multi-step xrun invocation:** You get the same behavior and feature set as the traditional direct invocation method using multiple commands (with three-step mode) to compile, elaborate, and simulate the design. Alternatively, you can use a two-step mode that allows you to compile and elaborate the design in the first step and then simulates the design in the second step.

Related Topics

- [xrun Command Syntax](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)
- [Overview of xrun](#)

xrun Command Syntax

You can invoke `xrun` using the following ways:

- **Single-step xrun invocation:** The most basic way to use `xrun` is to list the files that comprise the simulation on the command line, along with all command-line options. The syntax is:

```
xrun [options] <source_files>
```

For example:

```
% xrun -ieee1364 -v93 -access +r -gui verify.e top.v middle.vhd sub.v
```


- **Multi-step XRUN invocation:** When using the *xrun* utility in multi-step invocation, Xcelium supports a flexible use model with the same behavior and feature set as traditional direct invocation. You can run the following options to perform the various steps independently:

- **Compilation:** % `xrun -compile [options] <source_files>`
- **Elaboration:** % `xrun -elaborate [options] <source_files>`

Alternatively, you can use the two-step mode to compile and elaborate the design by using `xrun -c [options] <source_files>`.

- **Simulation:** % `xrun -R`

For AMS simulation, the `xrun` command recognizes the following source file types (`sourceFiles`) by their file extensions:

Source File Type	Valid Extensions
Spectre or SPICE	<code>.scs</code> or <code>.sp</code> You can specify additional extensions for analog source files using <code>-spice_ext</code> extension.
Verilog-AMS	<code>.vams</code> , <code>.VAMS</code> You can specify additional extensions for Verilog-AMS source files using the <code>-amsvlog_ext</code> extension.
SV-AMS	<code>.svams</code>
VHDL-AMS	<code>.vha</code> , <code>.VHA</code> , <code>.vhams</code> , <code>.VHAMS</code> , <code>.vhms</code> , <code>.VHMS</code> You can specify additional extensions for VHDL-AMS source files using <code>-amsvhdl_ext</code> extension.

For complete information about the file extensions that are recognized by `xrun` and how to change the default set, see [Customizing XRUN](#). You can also type `xrun -helpfileext` to see a list of file extensions.

You must not specify `-ams` on the `xrun` command line unless you want to force `xrun` to compile all Verilog and VHDL input files as AMS files.

While you can compile both SystemVerilog and AMS source files on the same command line, you cannot have a file that contains both SystemVerilog and AMS statements. See [Using SystemVerilog Modules](#).

Related Topics

- [Using xrun with Spectre and SPICE Input Files](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)
- [XRUN Options](#)

xrun Command-Line Options for Mixed-Signal Designs

The `xrun` command options listed in this topic are the commonly used options for analog and digital mixed-signal design simulations. For the complete set of `xrun` command options, see [Common Options](#).

To see the set of command-line options (`xrunOptions`) that relate to AMS simulation, type the following on the command line:

```
xrun -helpsubject ams
```

You can use the `xrun -helpsubject amsspice` to view the `xrun` options for AMS-SPICE.

xrunOptions	Description
-ams	Force <code>xrun</code> to compile all input files as AMS files (Verilog-AMS, VHDL-AMS) regardless of their file extensions. This option indicates that some or all of the HDL design units are written in the Verilog-AMS language. If you do not use this option, <code>xrun</code> analyzes for the Verilog language, which is likely to result in many errors when the language is actually Verilog-AMS. For example, to compile the files <code>ms10.v</code> and <code>ms12.v</code>, which both contain modules written with Verilog-AMS, you can use a command like <pre>xrun -ams ms10.v ms12.v</pre> Be careful to use the <code>-ams</code> option only when appropriate. For example, using the option with legacy digital Verilog modules can cause errors if Verilog-AMS keywords are used as identifiers in the Verilog modules.
-access[+] [-] access_spec	Sets the visibility access for all objects in the design.
-ade	Enable plotting of digital signals in ViVA on digital simulation in ADE flow.
-afile access_file	Specifies an access file.

<code>-amsvivalog</code>	Generate <code>runObjFile</code>. This is useful for several analyses supported by AMS, such as monte carlo analysis that creates multiple log files. In cases where multiple log files are saved in different locations, it is not possible for ViVA to read all the datasets unless there is a <code>runObjFile</code> that tells ViVA where to get all the log files from.
<code>-amsconnrules</code> <code>rulesName</code>	Specify connect rules to use for automatic connect module insertion (for example, between the default <code>logic</code> discipline and the <code>electrical</code> discipline) You can specify more than one <code>-amsconnrules</code> option. The order in which you specify connect rules in their source files determines their precedence. The software must be able to find the named connect rules in one of the source files or precompiled libraries. If you use the Cadence-installed connect rules, you can use an <code>ie</code> statement to automate the process of creating a custom discipline and connect rule for connecting the custom discipline to the <code>electrical</code> discipline.
<code>-amsdebug</code>	Generates readable call stack for the log file.
<code>-ams_dig_wreal</code>	Keeps <code>wreal</code> as discrete discipline.
<code>-ams_dspf_bbox</code>	Enables the DPSF-in-the-middle flow for AMS simulations in Virtuoso ADE.
<code>-amsinput spectre_file</code>	Specifies a Spectre-language file. You can specify more than one Spectre-language file on the <code>xrun</code> command line: <pre>xrun -amsi file1.scs -amsi file1.scs</pre>
<code>-amsoutdir</code>	Redirect the temporary files and subdirectories created during compilation, elaboration, and simulation to the specified subdirectory . The <code>-amsoutdir</code> option will override the <code>+amsrawdir</code>, <code>-nclibdirpath</code>, or <code>-ahdllibdir</code> settings. It is recommended not to use the <code>+amsrawdir</code>, <code>-nclibdirpath</code>, or <code>-ahdllibdir</code> options when using <code>--amsoutdir</code>.

<code>-amspartinfo file</code>	Specifies a file to hold mixed-signal partition and connect module insertion information. The file also contains information about Real Number Modeling nettype (for example, <code>wrealavg</code> , <code>wrealsum</code> , and so on) in the design. The format of the file might change from release to release.
<code>-amsdir raw_dir</code>	Specifies the output raw file directory.
<code>-amselabtrace assert digitaloomr sso veride</code>	Allows to trace nets in a design. You can specify the following values: • <code>assert</code>: Traces nets that are connected to digital constructs in an assertion statement. • <code>digitaloomr</code>: Traces OOMR nets that are used in digital context which result in a discrete discipline resolved. • <code>ssoveride</code>: Generates an error message containing the information about the hierarchical instance port in the digital blocks, if the supplySensitivity attributes are not found in that port when supply-sensitive connect modules are used.
<code>-amsvhdl_ext extension</code>	Overrides extensions for VHDL-AMS source files.
<code>-amsvlog_ext extension</code>	Overrides extensions for Verilog-AMS source files. You can also add file extensions to the list of built-in extensions, by specifying a plus sign (+) before the extensions to be added. For example, the following option adds <code>.va</code> to the list of built-in file extensions for Verilog-AMS source files: <pre>-amsvlog_ext +.va</pre>
<code>-ams_ucm</code>	Sets the Universal Connect Module (UCM) to be enabled by default for APS and Spectre X simulation performance mode.
<code>-ams_weak_setd</code>	Allows to override the BDR setting on domainless nets to any appropriate discipline.

<code>-analogcontrol file</code>	Specifies the analog simulation control file. The analog simulation control file is an ASCII text file written in the Spectre or Spectre control languages. The contents of the file control the behavior of the analog solvers. For example: <pre>xrun top:amsSS -cdslib /SAR_A2D/cds.lib -hdlvar /SAR_A2D/hdl.var -analogcontrol /SAR_A2D/tutorial_run/amsSC.scs</pre>
<code>-analogsolver</code>	Lets you specify how to run Spectre. Example: <pre>xrun -analogsolver "spectre"</pre> Runs Spectre on the local host. You can also use a script or a resource scheduler, provided it boots Spectre. <pre>xrun -analogsolver "bsub -R \"OSNAME==Linux &7 OSREL==EE70 && SFIPLATFORM==c0700\" -n 1 -q lnx64 -J spectre"</pre> Runs Spectre on a remote host.
<code>-anno_simtime</code>	Enables the use of PLI / VPI routines that modify delays at simulation time.
<code>-append_log</code>	Appends log data from multiple runs of <code>xrun</code> to one log file.
<code>-binding[lib .] cell [: view]</code>	Forces an explicit submodule binding.

<code>-cda_implicit_tmpdir</code> <code>implicitTmpDir</code>	Writes the compiled design data to <code>implicitTmpDir</code>, the implicit TMP directory for all libraries in the design. Establishes a directory to hold any created temporary libraries. This directory is used even when <code>ASSIGN</code> statements in the <code>cds.lib</code> file specify different directories. For example, your <code>cds.lib</code> contains <pre>DEFINE amstestLib ./amstest DEFINE basicLib ./basic ASSIGN basicLib TMP ./basicTMP DEFINE analogLib ./analog ASSIGN AllLibs TmpRootDir ./myTMPs</pre> Without a <code>-cda_implicit_tmpdir</code> option in effect, new data is written to <pre>./myTMPs/amstest ./basicTMP/basic ./myTMPs/analog</pre> Using the same <code>cds.lib</code> but with the option, <code>-cda_implicit_tmpdir ./myImplTMPs</code> in effect, however, writes new data to: <pre>./myImplTMPs/amstest ./myImplTMPs/basic ./myImplTMPs/analog</pre>
<code>-cda_implicit_tmponly</code>	When you use this option with the <code>-update</code> option, the update operation looks only at design data in the <code>implicitTmpDir</code> you specify using the <code>-CDS_IMPLICIT_TMPDir</code> option. If you do not use this option, the update operation also considers design data it finds in libraries defined in <code>cds.lib</code> files. You can use the <code>-CDS_IMPLICIT_TMPOnly</code> option only when you also use the <code>-CDS_IMPLICIT_TMPDir</code> option.
<code>-cdslib cdslib_pathname</code>	Specifies the <i>cds.lib</i> file to use.
<code>-chkdigdisp</code>	Performs discipline compatibility of the digital net.
<code>-checktasks</code>	Checks that all \$tasks are predefined system tasks.

<code>-clean</code>	Deletes the complete set of output files and directories that are created by the tool to perform additional processing of Verilog and SPICE interfaces. These files include the <code>portbind</code> files, skeletons, and other AMS Designer-specific processing files.
<code>-cleanlib</code>	Deletes all <code>.pak</code> files found in the libraries specified in the <code>cds.lib</code> file that is located in the current working directory or the <code>cds.lib</code> file specified using the <code>-cdslib</code> option. It also removes the <code>./xcelium.d</code> directory. The <code>-cleanlib</code> option searches through the entire <code>cds.lib</code> structure and also deletes any additional <code>cds.lib</code> files that are included in the original. This option does not remove the <code>.pak</code> files that do not have write permissions, or the <code>.pak</code> files located in the Cadence Install directory.  Use the <code>-cleanlib</code> option carefully because it deletes all the writable <code>.pak</code> files found in the libraries specified in the <code>cds.lib</code> file even if the libraries are shared with another process.
<code>-cleanlibverify</code>	Behaves similarly as the <code>-cleanlib</code> option. However, when this option is used, <i>xrun</i> displays the list of files that are being removed and removes the files only after confirmation.
<code>-cleanlibscript</code>	This option creates an executable script, which upon executing, produces the same result as the <code>-cleanlib</code> option. The script is called <code>cleanlibscript.sh</code> and is created in the working directory. After creating the script, <i>xrun</i> exits.
<code>-coverage</code>	Enables coverage instrumentation.
<code>-default_spice_oomr</code>	Assigns a default value (<code>1'bX</code>) when a digital statement contains an out-of-module reference to a SPICE block. See Reusing Mixed-Language Testbenches.
<code>-define identifier [= value]</code>	Defines a macro. See Defining Macros on the Command Line .

<code>-delay_mode{zero unit path distributed none}</code>	Specifies the delay mode to use for digital Verilog-AMS portions of the hierarchy.
<code>-desktop</code>	Specifies that you want to use the desktop simulator after starting the launch program.
<code>-discipline disciplineName</code>	Specifies the discipline to use for undisciplined digital wires.
<code>-disres default detailed none</code>	Sets discipline resolution.
<code>-dms_report</code>	Provides a high-level report of mixed-signal and co-simulation designs. It helps you understand the various design configurations used in a design. See Mixed-Signal Design Debugging Reports .
<code>-dms_perf</code>	Enables performance optimizations in some mixed-signal design.
<code>-epulse_neg</code>	Filters cancelled events (negative pulses) to the e state.
<code>-epulse_noneg</code>	Does not filter cancelled events (negative pulses) to the e state.
<code>-epulse_ondetect</code>	Uses On-Detect filtering of error pulses.
<code>-epulse_onevent</code>	Uses On-Event filtering of error pulses.
<code>-errormax integer</code>	Specifies the maximum number of errors to process.
<code>-expand</code>	Expands all vector nets.
<code>-fast_recompilation</code>	Enables enhanced recompilation, which improves recompilation and re-elaboration performance when compared to the default flow. When working with a larger design, or a design that requires temporary access to distinct source files, the default recompilation flow can reduce simulator performance or introduce unnecessary recompilation and re-elaboration. Use this option to enable the enhanced recompilation flow, which improves performance and helps to avoid any unnecessary recompilation and re-elaboration introduced by the default flow. You can specify this option on single or multiple xrun command lines. For more information, see the Specifying Enhanced Recompilation section in the <i>Xcelium Xrun User Guide</i>.

<code>-file arguments_filename</code>	Specifies a file of command-line arguments to use.
<code>-gateloopwarn</code>	Detects zero-delay loop of gate-level Verilog (or VHDL) models and prints the details in the form of proper error messages.
<code>-genafilename access_filename</code>	Generates an access file with the specified file name.
<code>-generic generic_name => value</code>	Specifies a value for a top-level generic.
<code>-hdlvar hdlvar_pathname</code>	Specifies the <i>hdl.var</i> file to use.
<code>-help</code>	Displays a list of the <i>xrun</i> command-line options.
<code>+finish_current_analysis</code>	Specifies <code>\$finish</code> to finish the current analysis for a multi-analysis simulation.
<code>-honorvams</code>	Specifying the <code>-honorvams</code> option with the <code>-sv</code> option ensures that a VAMS (Verilog-AMS) file extension is considered as a VAMS file and not compiled to a SystemVerilog (SV) file. It overrides all instances of the <code>-sv</code> option, which are read in multiple <code>.f</code> files, for these VAMS files. For example, in the following scenario, the <code>-sv</code> option compiles <code>foo1</code>, <code>foo2</code>, <code>foo3</code> as SystemVerilog files: <pre>> xrun foo1.sv foo2.vams foo3.v -sv</pre> To compile <code>foo2.vams</code> as VAMS, add the <code>-honorvams</code> option along with the <code>-sv</code> option: <pre>> xrun foo1.sv foo2.vams foo3.v -sv -honorvams</pre>
<code>-ieee1364</code>	Reports errors according to the IEEE 1364 Verilog standard.

<code>-iereport/-ieinfo</code>	The <code>-iereport/-ieinfo</code> option generates a detailed report containing Interface Element information, Port Discipline, Sensitivity information, Port Drivers information, Conversion Element (CE) name, File, Instance, Generic map, VHDL signal, SPICE node, CE report summary, and so on. See Mixed-Signal Design Debugging Reports. The <code>-ieinfo</code> option writes the results in a file <code>ams_ieinfo.log</code>. Use the <code>-ieinfo_log</code> option to output the results to a different file. The <code>-iereport</code> option is aliased to the <code>-ieinfo</code> option.
<code>-ieinfo_driverload</code>	Generates a Tcl file, <code>ieinfo_driverload.tcl</code> , with the drivers and loads information for both CE and IE.
<code>-ieinfo_driverload_tcl <filename></code>	Generates a user-specified Tcl file that contains drivers and loads information for both CE and IE.
<code>-ieinfo_log <filename></code>	Specifies the file to which the results of the <code>-iereport</code> are written. If this option is not specified, the results are written to the default <code>ams_ieinfo.log</code> file in the current working directory.
<code>-ieinfo_probe</code>	Generates a Tcl file <code>ieinfo_probe.tcl</code> that contains the probe-related information for both IE and CE.
<code>-ieinfo_probe_tcl <filename></code>	Generates a user-specified Tcl file that contains the probe-related information for both IE and CE.
<code>-ieinfo_summary</code>	Generates a summarized report containing IE-related information. See Mixed-Signal Design Debugging Reports. The <code>-ieinfo_summary</code> option writes the results in a file <code>ams_ieinfo.log</code>. Use the <code>-ieinfo_log <filename></code> option to output the results to a different file.

<code>-ignore_analog_tstop</code>	Ignores the <code>tstop</code> setting in the Spectre netlist during an AMS simulation. This ensures that the simulation continues running even after the transient time has exceeded the stop time. When this option is set, the stop time is controlled by the digital simulator. The stop time specified in the Spectre <code>tran</code> statement is still used to initiate the analog solver. For example, the matrix and time step.
<code>-ignore_missing_spice_port</code>	Ignores the missing SPICE ports displayed as "not found" in the port bind file.
<code>-ignore_spice_oomr</code>	Ignores any digital statements that contain out-of-module references to SPICE blocks See Reusing Mixed-Language Testbenches for more information.
<code>-ignore_svbind_spice</code>	Ignores the SV bind statements on SPICE blocks. See SystemVerilog Binding on SPICE for more information.
<code>-incdir directory</code>	Specifies an include directory.
<code>-intermod_path</code>	Enables multisource and transport delay behavior with pulse control for interconnect delays.
<code>-lexpragma</code>	Enables processing of lexical pragmas.
<code>-libcell</code>	Marks all cells with <code>`celldefine</code>.
<code>-libverbose</code>	Displays messages about module and UDP instantiations.

<code>-licinfo</code>	Provides information on the simulation-time licenses required for the specified design. It also provides additional information about the features that are being used for AMSD licensing. The <code>-licinfo</code> option generates a report on the licensing requirements using the following parameters: • The contents of the design (AMS, SystemC, SystemVerilog, Verilog, VHDL, and so on). • Other options specified on the command line. • Any arguments specified using the <code>-uselicense</code> option. The <code>-licinfo</code> option does not check out any licenses and also does not run the simulation.
<code>-linedebug</code>	Enables line debug capabilities.
<code>-loadpli1</code> <code>shared_library_name :</code> <code>bootstrap_function_name</code>	Dynamically loads the specified PLI 1.0 application.
<code>-loadvpi</code> <code>shared_library_name :</code> <code>bootstrap_function_name</code>	Dynamically loads the specified VPI application.
<code>-log_amsspice <log file></code>	Saves the <code>amsspice</code> output file into the specified log file.
<code>-logfile filename</code>	Specifies the file to contain log data.
<code>-maxdelays</code>	Applies the maximum delay value from a timing triplet in the form <code>min:typ:max</code> in the SDF file while annotating to Verilog or to VITAL.
<code>-mccodegen</code>	Enables multi-core code generation for AMS designs and RNM designs. See <code>-mccodegen</code>. Analog mixed-signal designs are not supported.
<code>-messages</code>	Turns on the printing of informative messages.

<code>-mixed_bus_opt</code>	Does not allow mixed buses to be automatically generated for unsupported constructs.
<code>-mixesc</code>	Required when elaborating if you instantiate VHDL or VHDL-AMS in a Verilog or Verilog-AMS module and you use escaped entity, port, or generic names within the VHDL or VHDL-AMS descriptions.
<code>-modelincdir pathname</code> <code>{: pathname}</code>	Specifies a list of paths to be searched for model files, included files, or files that are passed as instance parameter values.
<code>-modelpath string</code>	For Verilog-AMS, specify one or more model files, optionally including a model section specifier such as <code>-modelpath ./models/resistor.scs(res)</code>
<code>-neg_tchk</code>	Allows negative values in <code>\$setuphold</code> and <code>\$recrem</code> timing checks in the Verilog description and in <code>SETUPHOLD</code> and <code>RECREM</code> timing checks in SDF annotation.
<code>-nettype_port_relax</code>	Allows for relaxed port compatibility rules for connections of built-in nettypes. See Using Real Number Modeling in SystemVerilog .
<code>-neverwarn</code>	Disables printing of all warning messages.
<code>-no_sdfa_header</code>	Turns off the printing of elaborator information messages that display information contained in the SDF command file.
<code>-no_tchk_msg</code>	Turns off the display of timing check warning messages.
<code>-no_tchk_xgen</code>	Turns off X generation in accelerated VITAL timing checks.
<code>-no_vpd_msg</code>	Turns off glitch messages from accelerated VITAL pathdelay procedures.
<code>-no_vpd_xgen</code>	Turns off X generation in accelerated VITAL pathdelay procedures.
<code>-noautosdf</code>	Turns off automatic SDF annotation.
<code>-nocopyright</code>	Suppresses printing of the copyright banner.
<code>-noipd</code>	Turns off recognition of input path delays in a VITAL level 1 cell and uses the non-delayed input signals directly.
<code>-noline</code>	Turns off source line locations for errors.
<code>-nolog</code>	Turns off generation of a log file.

<code>-nonotifier</code>	Tells the elaborator to ignore notifiers in timing checks.
<code>-nomempack</code>	Enables use of the PLI routine <code>tf_nodeinfo()</code> , which is used to access memory array values.
<code>-noparamerr</code>	Tells the elaborator to allow undeclared parameters to be overridden.
<code>-noporterr(scope) port_name</code>	Tells the elaborator to allow the instantiation of design units that do not have all the ports that are specified in the port connection list. For more information, see the <code>-noporterr</code> option.
<code>-nopragmawarn</code>	Disables pragma-related warning messages.
<code>-nosource</code>	Turns off source file timestamp checking when using the <code>-UPdate</code> option.
<code>-nostdout</code>	Suppresses the printing of most output to the screen.
<code>-notimingchecks</code>	Turns off the execution of timing checks.
<code>-novitalaccl</code>	Suppresses the acceleration of VITAL level 1-compliant cells.
<code>-nowarn warning_code</code>	Disables printing of the specified warning message.
<code>-ntc_warn</code>	Prints convergence warnings for negative timing checks for both Verilog and VITAL if delays cannot be calculated given the current limit values.
<code>-nowarn <arg></code>	Disables printing of the specified warning message.
<code>-noparamerr</code>	Disables flag setting undefined parameters as error.
<code>-no_analog_solver</code>	Runs the simulation without invoking the Spectre AMS simulator in digital-only logic and RNM simulations. This option is automatically turned on when there is digital-only logic or RNM designs that do not have electrical/SPIICE or need ie card (.scs). Also, you can manually specify this option if you do not want to invoke Spectre AMS simulator in a design where ie card (.scs) is available.
<code>-omcheckinglevel checking_level</code>	Specifies the OMI checking level to use.

<code>-pathpulse</code>	Enables <code>PATHPULSE\$</code> specparams, which are used to set module path pulse control on a specific module or on specific paths within modules.
<code>-plinoptwarn</code>	Prints a warning message only the first time that a PLI read, write, or connectivity access violation is detected.
<code>-plinowarn</code>	Disables printing of PLI warning and error messages.
<code>-pragma</code>	Enables pragma processing.
<code>-preserve</code>	Preserves resolution functions on signals with only one driver.
<code>-propspath path</code>	Specifies the path to the <code>prop.cfg</code> file. However, <code>prop.cfg</code> is no longer supported; therefore, you must convert to amsd Block . See Appendix E: Migration of prop.cfg file to an amsd Block . Alternatively, you can instruct the internal translator to convert your <code>prop.cfg</code> file to an amsd block file, <code>prop.cfg.scs</code> , by setting the AMSCB environment variable.
<code>-ppe</code>	Invokes the post-processing environment (PPE) See Running the SimVision Analysis Environment for more information.
<code>-propspath path</code>	Specifies analog occurrence property database file
<code>-pulse_e error_percent</code>	Sets the percentage of delay for the pulse error limit for both module paths and interconnect.
<code>-pulse_int_e error_percent</code>	Sets the percentage of delay for the pulse error limit for interconnect only.
<code>-pulse_int_r reject_percent</code>	Sets the percentage of delay for the pulse reject limit for interconnect only.
<code>-pulse_r reject_percent</code>	Sets the percentage of delay for the pulse reject limit for both module paths and interconnect.
<code>-q</code>	Suppresses all informational messages (quiet mode).
<code>-relax</code>	Enables relaxed VHDL interpretation.
<code>-rnm_coerce default none detailed on off scopeType-scope-</code>	Enables or disables wreal coercion globally and/or on certain design scopes.

You can turn off wreal coercion:

- On a specific instance and instances under it.
- For instances whose master is specific module and instances under the module.
- On specific nets.

You can turn on wreal coercion:

- On a specific instance and instances under it.
- For instances whose master is specific module and instances under the module.

It is not supported on scopetypes: `lib`, `cellterm`, `instterm`, `net`, and `cellupport`.

Possible values are:

- `none`: Disables wreal coercion globally.
- `detailed`: Enables wreal coercion with detailed resolution. This is the default.
- `default`: Enables global coercion with default resolution.
- `on/off scopeType -scope-`: Enables wreal coercion on specific scope. You can use `on scopeType -scope` to enable wreal coercion on the specified scope, when coercion is disabled on other scopes. Likewise, you can use `off scopeType -scope` to disable wreal coercion on a specified scope, when coercion is enabled on other scopes.

- RNM (wreal) coercion on specific scope is not supported in incremental elaboration single xrun flow.
- When specifying the `scopeType -scope` value as `off`, you can specify the `-dms_perf` command-line option for performance optimization.

Programmable coercion off is supported on scopeTypes: `net`, `inst`, `instterm`, `cell`, `cellterm`, or `lib`. For example:

```
"NET- hierarchical_net_name -"  
"INSTTERM- hierarchical_port_name -"
```

```
"INST- hierarchical_instance_name -"  
"CELLTERM-1 ib_name.cell_name.port_name -"  
"CELLTERM- cell_name.port_name -"  
"CELL- lib_name.cell_name:view_name -"  
"CELL- lib_name.cell_name -"  
"CELL- cell_name -"  
"LIB- lib_name -"
```

Example:

```
rnm_coerce "off inst-top.dcinstant"
```

All the net of instance `top.dcinstant` and its children will not be coerced to wreal; top level and other instances will be coerced as normal.

- The functionality of `instterm` assignment has been merged into `net`, for the same scope. Therefore, if you specify both the assignments, the `net` assignment takes higher precedence. However, if the `net` assignment is not specified, the `instterm` assignment is considered.
- If `scopeType` is not specified, the behavior would be similar to the `none` option, and wreal coercion is disabled.
- Coercion of a wire is performed for the entire wire (bus). Coercion is not supported at bit-level for a bus.

If multiple `-rnm_coerce` options are specified for the same design scope, the one specified with the immediate design scope takes precedence.

If you are a digital-centric user running an AMS simulation that requires only the digital solver, it is recommended to specify `-rnm_coerce none`.

<code>-rnm_tech</code>	Enables Real Number Modeling (RNM) in Elaboration Mode. The option enables you to compile and elaborate structural netlists/file in SystemVerilog (.sv file) while applying the concepts of 'implicit' interconnects and RNM coercion; thereby, providing a behavior that is equivalent to structural netlists in Verilog-AMS (.vams files).
<code>-rnm_type_priority</code> <code><type_list></code>	Allows you to specify the precedence order for choosing the type of coercion for an interconnect net when it connects to different nettypes of ports. You can specify the multiple types, without space, separated with the greater than symbol (>), comma (,) or hyphen (-). For example: <pre>-rnm_type_priority wrealsum>wrealmax>wrealavg</pre> or <pre>-rnm_type_priority wrealsum,wrealmax,wrealavg</pre> or <pre>-rnm_type_priority wrealsum-wrealmax-wrealavg</pre> In this example, if a nettype of wrealsum is present in the connection, it gets priority and the wire is coerced as wrealsum. You must specify the <code>-rnm_tech</code> option along with the <code>-rnm_type_priority</code> option to enable coercion. See -rnm_tech.
<code>-scope_discipline</code> <code>disc_scope</code>	Specifies one scope-based discipline.
<code>-sdf_cmd_file</code> <code>sdf_command_file</code>	Specifies an SDF command file to control SDF annotation.
<code>-sdf_no_errors</code>	Suppresses error messages from the SDF annotator.
<code>-sdf_no_warnings</code>	Suppresses warning messages from the SDF annotator.
<code>-sdf_nocheck_celltype</code>	Disables celltype validation between the SDF annotator and the Verilog description.
<code>-sdf_precision</code> argument	Modifies SDF data to this precision.
<code>-sdf_verbose</code>	Includes detailed information in the SDF log file.

<code>-sdf_worstcase_rounding</code>	Truncates the min value, rounds the typ value, and rounds up the max value for the timing values in the SDF file.
<code>-setdiscipline[no_dr]</code> <code>disc_scope disc_name</code>	Specifies a discipline for the elaborator to use for domainless nets in the specified scope. Using <code>no_dr</code> turns off discipline resolution in the specified scope and for the entire block. And, the <code>no_dr</code> option cannot be used in a block that has nets with explicit discipline.
<code>-shm_filter_group</code> <code><precision></code>	Enables real value filtering for all real value probes. You can specify a precision (delta value) for real number probes to remove recorded values whose delta is less than the precision specified. For more information, see Enabling Real Value Probe Filtering .

`-simcompatible_ams
compat_val`

Specifies whether to set simulation values for compatibility with the Spectre language or with the HSPICE language. The default is `hspice`.

Specifies the compatibility setting for simulation values. Valid settings are as follows:

- `spectre`: Spectre-compatible simulation values
- `hspice`: HSPICE-compatible simulation values

The default is `hspice`. The simulation values are as follows:

`-simcompatible_ams hspice`

- `tnom` and `temp` are set to 25C
- Parameter inheritance is set to global, so that global parameter definitions override local ones.
- Forces IC statements and initial conditions on elements for DC and OP analyses.

`-simcompatible_ams spectre`

- `tnom` and `temp` are set to 27C
- Parameter inheritance is set to local only
- Forces initial conditions only when you set the DC analysis `force` option

In addition, for `-simcompatible_ams hspice`, flags on all device models are set to be SPICE compatible.

For example:

```
xrun -simcompatible_ams hspice top:amsSS
```

The simulator uses these values regardless of which language you use.

The `-simcompatible_ams` option does not affect parsing. If you want to add SPICE commands in model files or in the analog simulation control file, you must use the statement,

```
simulator lang=spice
```

to specify the language for the statements that follow.

<code>-snapshot snapshot_name</code>	Specifies a name for the simulation snapshot. If you do not use this option, <code>xrun</code> places the snapshot in the view directory of the first design unit specified on the command line, even when the first design unit is a configuration.
<code>-solver spectre aps</code>	Specifies whether the Spectre solver, or the APS solver is to be used with the AMS Designer simulator. If this parameter is not specified with the <code>xrun</code> command, the Spectre solver is used by default.
<code>-specificunit[lib .] cell [: view] filename]</code>	Compiles the specified unit from the source file.
<code>-spectre_args</code>	Specifies one or more Spectre command-line arguments. You can also include multiple entries of the <code>-spectre_args</code> parameter on the command line, which are concatenated during command processing. Note: <code>-spectre_args</code> is ignored if APS solver is selected.
<code>-spectre_argfile_spp filename</code>	Specifies the path to a file containing space-separated command arguments for <code>spp</code> . You do not need this option if you are using the simulation front end (SFE) parser.
<code>-spectre_e</code>	Runs Spectre parser with the <code>-E</code> option
<code>-spectre_path</code>	This option is used in the AMSD flex mode to specify the directory where <code>spectre</code> and <code>spectre_root</code> are located. These can be original binaries from an installation directory or wrapper scripts provided by the user. <code>xrun</code> runs the <code>spectre_root</code> executable from the specified path to locate the Spectre installation and this location is used to run the simulations. Syntax: <pre>xrun -64... -spectre_path <path_to_spectre> ...</pre> Here, <code><path_to_spectre></code> can be the path to Spectre installation or path to a Spectre wrapper script. If <code>-spectre_path</code> points to a Spectre wrapper script, the following requirements must be met:

- The Spectre wrapper should be an executable.
- Wrapper for `spectre_root` should be provided at the same location as the Spectre wrapper.
- The `spectre_root` wrapper should also be an executable and it should only echo the output of the installed `spectre_root`.

Example of Wrapper Script Usage:

The following examples illustrate how `spectre` and `spectre_root` should be wrapped in a scenario where a specific user group is to be assigned to use Spectre 18.1 and all the other users are to use Spectre 19.1.

spectre wrapper

```
#!/bin/sh
if [ "$USER" == "<username>" ]
then
spectreInstallDir=/<spectre_installation_dir>/18.1_ISR13
else
spectreInstallDir=/<spectre_installation_dir>/19.1_ISR4
fi
$spectreInstallDir/bin/spectre "$@"
```

spectre_root wrapper

```
#!/bin/sh
if [ "$USER" == "<username>" ]
then
spectreInstallDir=/<spectre_installation_dir>/18.1_ISR13
else
spectreInstallDir=/<spectre_installation_dir>/19.1_ISR4
fi
$spectreInstallDir/bin/spectre_root "$@"
```

`-spectre_spp`

Runs the Spectre parser with `spp` on when parsing files specified by the `-MODELPath` option. You do not need this option if you are using the simulation front end ([SFE](#)) parser.

<code>-spectre_validated</code>	Lists the Spectre versions compatible with the current Xcelium release. Example: <pre>xrun -spectre_validated => Spectre 19.1.0.123.isr18</pre> To view the Xcelium versions validated with the current Spectre release, use the spectre option <code>-xcelium_validated_version</code>. Example: <pre>spectre -xcelium_validated_version => Xcelium 20.09-s016</pre>
<code>-spice_ext extension</code>	Overrides extensions for Spectre and SPICE source files. See Using xrun with Spectre and SPICE Input Files.
<code>-status</code>	Prints statistics on memory and CPU usage.
<code>-sysv_ext extension</code>	Compiles custom file extension as SystemVerilog file. For example, the following option compiles with a <code>.dvams</code> file extension as an SV file: <pre>-sysv_ext .dvams</pre>
<code>-top top_unit</code>	Specifies <code>top_unit</code> as the top-level design unit. You must use <code>-top</code> if the top level of your design is VHDL. Note: <code>xrun</code> automatically determines the top-level design unit from Verilog or SystemVerilog source files. See -top. For more information about using <code>-top</code> to specify connect modules and <code>cds_globals</code>, see "Compatibility with Existing Use Models".
<code>-timescale`time_unit / time_precision'</code>	Sets the default timescale for Verilog (digital) modules that do not have a timescale set.
<code>-typdelays</code>	Applies the typical delay value from a timing triplet in the form <code>min:typ:max</code> in the SDF file while annotating to Verilog or to VITAL.

<code>-unit[lib .] cell [: view]</code>	Specifies the unit to be compiled.
<code>-upcase</code>	Changes all identifiers (including keywords) to upper case (case-insensitive). Using this option can cause conflicts. For example, if you use this option, you must create and use a case-insensitive version of the <i>disciplines.vams</i> file that distinguishes <code>Voltage</code> nature from <code>voltage</code> discipline and <code>Current</code> nature from <code>current</code> discipline. In addition, using this option causes a clash between the <code>Force</code> nature and the <code>force</code> keyword.
<code>-update</code>	Recompiles out-of-date design units. The <code>-CDS_IMPLICIT_TMPOnly</code> option affects the behavior of the <code>-update</code> option.
<code>-uselicense mnemonic_list</code>	Specifies a custom prioritized license checkout order for simulation. For example: <pre>-uselicense mnemonicList [:DEFAULT]</pre> where <code>mnemonicList</code> is a colon-separated list of one or more valid mnemonic keywords. The optional <code>DEFAULT</code> mnemonic indicates the default license selection, which includes digital-specific license mnemonics. For more information about these and other digital-specific mnemonics, see Product and Licensing Information for Mixed-Signal Designs. The <code>-uselicense</code> option has been deprecated in the XCELIUM 16.11 release.
<code>-v93</code>	Enables VHDL-93 features.
<code>-version</code>	Prints the compiler version number.
<code>-view view_name</code>	Specifies a view association.
<code>-vipdmax</code>	Selects the Max. delay value for <code>VitalInterconnectDelays</code> .
<code>-vipdmin</code>	Selects the Min. delay value for <code>VitalInterconnectDelays</code> .

<code>+wf_alias</code>	Using this option enables the waveform aliasing feature, which improves the waveform database size by leveraging the alias semantics for waveform storage. The <code>+wf_alias</code> option would be removed in a future release.
<code>-work library</code>	Specifies the library to be used as the work library.
<code>-wreal_resolution resolutionFunction</code>	Specifies the <code>wreal</code> resolution function you want the elaborator to use; valid values for <code>resolutionFunction</code> are: • <code>default</code> - Default setting • <code>fourstate</code> - Verilog 4-state logic resolution algorithm • <code>sum</code> - Summation of all drivers • <code>avg</code> - Average of all drivers • <code>min</code> - Minimum value of all drivers • <code>max</code> - Maximum value of all drivers
<code>-wreal2vhdlmap file</code>	Maps VHDL analog X/Z states with the Verilog <code>wrealZ/X</code> states. You can specify the VHDL/<code>wreal</code> X and Z state mappings in a <code>file</code>. The template format is in the form of <code>wreal2vhdlmap</code> card which defines the mapping for the specified VHDL data type. Multiple <code>wreal2vhdlmap</code> cards can be specified in the file. See Mapping Analog X/Z States with Verilog <code>wrealZ/X</code> States..
<code>-wreal2vhd_probe</code>	Generates a TCL probe file named, <code>wreal2vhd_probe.tcl</code> file. It probes the Wreal and VHDL ports where <code>xrun</code> code implements the <code>wreal2vhdl</code> mapping.
<code>-zparse SKILL_file</code>	Enables zparsing. The <code>SKILL_file</code> argument specifies the name of the SKILL file this option creates for importing Verilog-AMS text modules into the Virtuoso® design environment.

Related Topics

- [XRUN Options](#)
- [xrun Command Syntax](#)

Examples Using xrun for AMS Simulation

You can specify different file types on the `xrun` command line. `xrun` uses the file extension to determine the file type, such as Verilog (`.v`), Verilog-AMS (`.vams`), and analog (`.scs`). For example, the following `xrun` command runs the AMS Designer simulation using the Spectre solver:

```
xrun top.v test1.vams test.scs
```

To specify an AMS Designer simulation using the APS solver, type the following `xrun` command:

```
xrun -solver aps top.v test1.vams test.scs
```

Related Topics

- [XRUN Invocation for Mixed-Signal Designs](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)

Using xrun with Spectre and SPICE Input Files

The `xrun` utility's support of Spectre and SPICE input file is consistent with what Cadence's SPECTRE technology offers (which applies to the Spectre circuit simulators). This use model enables you to re-configure some of the digital blocks in your existing design with their SPICE equivalents. This allows you to switch seamlessly between digital verification and mixed-signal verification.

`amsspice` is a Cadence AMS Designer internal executable that is used by `xrun` to process SPICE/analog blocks. When you specify a file with the extension `.sp` or `.scs` to the `xrun` command, it is automatically parsed and interpreted by `amsspice`.

- A `.sp` extension indicates a SPICE syntax input file.
- A `.scs` extension indicates a Spectre syntax input file.

If you use `-spice_ext` to change the extension to `.cir` or `.ckt` (for example), you must be sure to use the appropriate `simulator lang` assignments.

You need to specify the following information for AMSSPICE setup:

- Original HDL (Verilog, VHDL etc.) design file(s)
- Top-level analog file containing subcircuit definition(s), model file(s) and so on
- AMS configuration information on how SPICE and HDL interact, using an `amsd` block

Your top-level analog file can contains SPICE subcircuit definitions, analog models, analog simulation control statements, anything you can put in an analog simulation control file, and the amsd block. You can put all your analog simulation control statements in a single file that you include in this top-level analog file so you do not need to use the `-analogcontrol` option. You can specify other Spectre commands in the input file as long as you do not specify them in the amsd block. Other Spectre commands (such as transient analysis command) must not appear in the amsd block.

For example:

```
run all_spice.scs anyOtherInputFiles...
```

where `all_spice.scs` contains the following:

```
// all_spice.scs

include "source/analog/PLL.sp" // analog subcircuit definitions
include "analog_top.scs" // analog subcircuit definitions
include "nmos1.scs" // nmos models
include "pmos.scs" // pmos model
include "acf.scs" // analog control file

//amsd block

amsd{

    portmap subckt=pll_top autobus=yes    config cell=pll_top use=spice    ie vsup=1.8

}
```

Following is an example of AMSD control file containing SPICE information (using the include statement) including models information and multiple amsd blocks specified in different files:

```
// file1.scs

    include "s1.cir"    include "s2.cir"

amsd{
    portmap subckt=s1 autobus=yes    config cell=s1 use=spice
}

amsd{
    portmap subckt=s2 busdelim=_    config cell=s2 use=spice    ie vsup=3.5
}

// file2.scs

include "subcx.sp"

amsd{
    portmap subckt=subcx    config cell=subcx use=spice
}
```

You can use the following methods to introduce SPICE models into Verilog-AMS:

- Using the `amsd` block. For example:

```
include "analog_top.sp"

amsd{

    portmap subckt=analog_top file="analog_top.pb"
    config cell=analog_top use=spice

}
```

- Using the `-modelpath` option. With the introduction of `xrun use model`, this option is no longer mandatory.
- Using the `amsd_subckt_bind=yes` parameter in the `'include` statement in SPICE files. For example:

```
//analogmodel.scs

include "analog_top.cir" amsd_subckt_bind=yes
```

`amsd_subckt_bind` is a flag to indicate that the SPICE file is `xmelab`-aware to bind an instance to SPICE. This is equivalent to specifying the `-modelpath <modelfiles>` option. Without the `amsd_subckt_bind` option, `xmelab` could not get the SPICE list, and can generate an error.

Related Topic

- [xrun Command-Line Options for Mixed-Signal Designs](#)

Migrating to Single-Step XRUN Invocation

You can run the AMS Designer simulator and the Xcelium simulator with the mixed-signal option using *xrun* in single-step mode or using the traditional direct invocation (three-step method) use model. It is recommended that you use *xrun* single-step mode for mixed-signal design verification. The following information can help you migrate from direct invocation mode to using single-step *xrun*.

File/Information for Three-Step	...for xrun
<code>cds.lib</code> (required)	Not required (remove or convert to <code>xrun</code> options)
<code>worklib</code> (required)	Not required (remove or convert to <code>xrun</code> options)
<code>hdl.var</code> (optional)	Not required (remove or convert to <code>xrun</code> options)
connect modules and connect rules	Use the <code>ie</code> statement in an <code>amsd</code> block

model path	Include device model files, optionally with a model section specifier, in a control file along with your <code>amsd</code> block For example: <pre>* AMS control file -- amsdControl.scs include "./source/design.scs" //analog netlist include "./models/model.scs" //device model files include "./models/diode.scs" section=dio // diode.scs is the model file; "dio" is the section to use include "./models/pmos1.scs" section=nom // pmos1.scs is the model file; "nom" is the section to use include "./analogControl.scs" //analog control file //amsd blockamsd{ portmap subckt=pll_top autobus=yes config cell=pll_top use=spice ie vsup=2.0 }</pre>
-propspath prop.cfg	Specifying a <code>prop.cfg</code> is no longer supported; therefore, you must convert to using an <code>amsd</code> block. See Migrating to an amsd Block from prop.cfg. Alternatively, you can instruct the internal translator to convert your <code>prop.cfg</code> file to an <code>amsd</code> block file, <code>prop.cfg.scs</code>, by setting the AMSCB environment variable.
Tcl probes	No change
analog control file	Specify the analog control file directly on the <code>xrun</code> command line; you do not need to use the <code>-analogcontrol</code> option; include one or more <code>amsd</code> blocks in this file.
discipline	Optional; if you do not specify a discipline (using the <code>-discipline</code> command-line option to <code>xrun</code> Command Syntax), the default discipline is logic ; see also the mode specifier on the <code>ie</code> statement.
timescale	No change
-ams option (required)	Not required (remove); <code>xrun</code> recognizes AMS input files by their extensions.

Here is an example of a three-step example that you can simplify using `xrun`:

Direct Invocation example	Single-step xrun equivalent
<pre> xmvlog -ams \ -cdslib cds.lib \ ./source/digital/*.v xmclab worklib.testbench \ connectLib.ConnRules_18V_full \ -cdslib cds.lib \ -hdlvar hdl.var \ -snapshot top:snapshot \ -amsf \ -timescale 1ns/100ps \ -discipline logic \ amsd.scs \ xmsim top:snapshot \ -cdslib cds.lib \ -analogcontrol top.scs \ -input probe.tcl \ -simcompat hspice </pre>	<pre> xrun ./source/digital/*.v \ ./amsf.scs \ -amsf \ -timescale 1ns/100ps \ -input probe.tcl </pre>

Related Topics

- [amsd blocks](#)
- [-analogcontrol](#)
- [-discipline](#)
- [xrun Command Syntax](#)
- [Xcelium System Tasks](#)

Mixed-Signal Design Debugging Reports

The following table lists the `xrun` command-line options and the various reports that you can generate for debugging purposes.

`-dms_report`

Provides a high-level report of mixed-signal and co-simulation designs. It helps you understand the various design configurations used in a design such as AXUM SPICE, electrical, Verilog-AMS, VHDL-AMS, SV-AMS, SV interconnect, SV-UDN, \$cged module, Inherited connections, IE card, coercion, autospiceoomr, \$real_net_alias, RNM Randomization, RNM Coverage, mixed-signal array of instances (AOI), VHDL + SPICE flow, SV Interface Interact, SV Bind Interact, SV Logic variable/Real variable to Electrical, Electrical IE, Built-in IE, SV UDN IE, Testbench Reuse (Virtuoso IP or SV) using DMS binding, and AMS UNL using DMS binding.

The following is an example of a report generated using `-dms_report`:

TOOL: xmelab 22.08-a071-20220815

DMS Design Report generated for 'worklib.top:vams':

The following mixed signal related content was processed (high level report):

Mixed Signal AOI: YES

SPICE files: YES

Verilog-AMS files (with electrical): YES

Verilog-AMS files (with wreal generic): YES

RNM randomization: YES

RNM coverage: YES

Coercion counts:3

Autospiceoomrs counts:1

If the elaboration terminates unexpectedly, the report generated may be incomplete and a note would be included in the report indicating this.

`-iereport/-ieinfo`

The `-iereport/-ieinfo` option generates a detailed report containing Interface Element information, Port Discipline, Sensitivity information, Port Drivers information, Conversion Element (CE) name, File, Instance, Generic map, VHDL signal, SPICE node, CE report summary, and so on. For example:

```
Interface Elements at the block <instance> testbench.msbuf.I2 of
<master> ana_nand (file :
/home/bcui/BDR_multpwr/source/analog/ana_nand.vams)
Automatically inserted : testbench.msbuf.I2.Y1__E2L__logic18V
Connect Module : E2L
Mode : Merged
Net : testbench.msbuf.I2.Y1 (discipline: discrete_amslps,
nettype: wrealsum)
Port :
testbench.msbuf@ms_buf<module>.I2@ana_nand<module>.I1@my_inv<module>.in
(logic18V input)
Parameters:
vsup : 1.8
List of Ports connected to net testbench.msbuf.I2.Y1 : (Total: 1)
testbench.msbuf.I2.I1.in (logic18V input)
Discipline of Port (Din): discrete_amslps, Wreal Port
Drivers of Port Din: tb.dut.PMUInst) assign wvdd1= vsup1
IEEE1801 Power Supply net
CE #1: Name: MY_AD_LIB.E2ILOG:behavior
File: E2ILOG.vhms
Instance: :vh_top:test1:e2ilog_a
Generic Map: ()
VHDL Signal: output ':vh_top:A' with type 'ilog'
Spice Node name: dummy_spice.A
---- CE Report Summary:
E2ILOG ( ELECTRICAL inout; ILOG out; ) total: 1
ILOG2E ( ILOG in; ELECTRICAL inout; ) total: 1

The -ieinfo option writes the results in a file ams_ieinfo.log. Use the -
ieinfo_log option to output the results to a different file.
```

The `-iereport` option is aliased to the `-ieinfo` option.

```
-ieinfo_summary
```

Generates a summarized report containing IE-related information. For example:

```
1. Interface Elements at the block <instance> top of <master> top
(file : ./top.sv)

Automatically inserted : top.\b_zero_pad_bit0__R2E_2__electrical

Connect Module : R2E_2
Mode : Merged
Net : pad connection (discipline: logic, nettype: variable)
Port : top.ana_gate@analog_top<module>.\itunes[1] (discipline:
electrical, direction: input, nettype: electrical)

Parameters :
vsup :5

          vd elta :0.078125
          vlo :0
          vx :0
          tr :5e-11
          tf :5e-11
          ttol_t :5.0000000000000001e-12
          tdelay :0
          rout :200
          rx :200
          rz :10000000

Discipline of Port (Din): logic, Wreal Port
Discipline of Port (Aout): electrical, Analog Port

Sensitivity information:
No Sensitivity info

IE Report Summary (with disciplines and directions):
R2E_2 ( logic input; electrical output;) total: 3

-----
---
```

Effective Number of IE Instances:
Total Number of Connect Modules : 3

The `-ieinfo_summary` option writes the results in a file `ams_ieinfo.log`. Use the `-ieinfo_log <filename>` option to output the results to a different file.

Related Topic

- [xrun Command-Line Options for Mixed-Signal Designs](#)

PSpice Netlist and Device Models

AMS Designer, with Spectre solver family (Spectre, Spectre APS, and Spectre XPS Mixed-Signal (MS) mode), supports the PSpice[®] netlist format targeted to include PCB components modeled in the PSpice format into a Spectre integrated circuit simulation. This solution does not support PSpice-only designs. The top-level netlist and control statement need to be defined in Spectre or SPICE format. The recommended approach is to define a subcircuit in PSpice netlist format and instantiate the subcircuit in the Spectre netlist.

For the AMS Designer simulator to support PSpice netlists, use the `+mts` argument with the `-spectre_args` option of the `xrun/xmsim` command:

```
xrun -spectre_args +mts
```

It is mandatory to specify the `+mts` argument in AMS to support PSpice netlists.

After you set `-spectre_args +mts`, you can include a PSpice netlist in your test bench database, through a Spectre netlist by using the following statement.

```
pspice_include < file > (Spectre format)
```

```
.pspice_include < file > (SPICE format)
```

All content of the included `file` should be in PSpice format. If the file includes additional files, they also should be in PSpice format. Elements and device models used in the PSpice netlist are simulated using PSpice default values and equations.

The Spectre PSpice feature does not support digital PSpice elements and PSpice encryption. Spectre encryption is recommended, if encryption is required

Related Topic

- [xrun Command-Line Options for Mixed-Signal Designs](#)

License Queuing

To turn on the queuing-for-license capability, you can use the `+lqtimeout <value>` argument. Specify the value in seconds to set how long to wait for a license. Value `0` means wait until the license is available. You might use `+lqt` as an abbreviation of `+lqtimeout`.

For example, the following command instructs the tool to wait for analog solver licenses until they are available.

```
xrun -spectre_args "+lqt 0"
```

The `+lqsleep <value>` argument enables you to set the sleep time between two attempts to check out a license when queuing. Setting the value to a positive number overrides the default sleep time of 30 seconds. You might use `+lqs` as an abbreviation of `+lqsleep`.

For example, the following command instructs the tool to check for the availability of analog solver licenses every 5 minutes.

```
xrun -spectre_args "+lqs 300"
```

The `+lsuspend` argument (applied by default) allows you to suspend or resume the license for Spectre during the simulation run. However, you can use the `-lsuspend` argument to disable this feature. In other words, `-lsuspend` is equivalent to `-nolicsuspend` on the digital side.

Standalone Spectre does not have `+lsuspend` by default; this is an AMS-only behavior. This command will have an effect only on the analog licenses.

Related Topics

- [xrun Command-Line Options for Mixed-Signal Designs](#)
- [AHDL Linter](#)

AHDL Linter

You can use the `-ahdllint` command-line option to turn on the AHDL linter feature that enables you to detect modelling issues in analog/mixed-signal Hardware Description Languages (AHDL). The AHDL linter feature comprises of static and dynamic lint checks. Static lint checks are performed before analysis. Dynamic lint checks are performed during analysis for dynamic modelling issues.

Dynamic lint checks can also be performed on VHDL-AMS modules.

Possible values for the `-ahdllint` option are:

`no` - Disables lint checks. There is no change in the existing compilation or simulation warning messages.

`warn` (default)- Turns on the static lint and dynamic lint checks. Except for the models with attribute (`-ahdllint = no`), the static linter checks all models, continues the simulation, and then performs dynamic lint checks.

`error` - Turns on the static lint check. Dynamic lint checks are performed only when static lint issues are not detected. Except the models with attribute (`-ahdllint = no`), the static linter checks all models.

The simulator generates an error and exits if there is any static lint warning reported after parsing all the models of the circuit. If there are no static lint warnings, the simulator continues the simulation and performs dynamic lint checks. However, in the case of dynamic lint issues, the simulator does not error out.

`force` - Similar to `warn`, but this option overrides the model attribute `ahdllint = no`, and to check all models, continue the simulation, and perform dynamic lint checks.

You can use the `-ahdllint_maxwarn = n` command-line option to control the maximum number of static warnings generated per Verilog-A, Verilog-AMS, or VHDL-AMS model. The default value is 5. The `-ahdllint_maxwarn` option does not limit the warnings generated from the dynamic lint checks.

Use the `-ahdllint_log = file_name` command-line option to dump all AHDLLinter static and dynamic and summary messages to a file.

The lint checks are performed during `xmsim` stage.

Though linter checks are supported for both Verilog-A and VerilogAMS languages, there can be some differences in the behavior. For example, when a declared variable is not used in the model, Verilog-A will generate a lint warning but VerilogAMS will not.

Dynamic lint checks can also be performed on VHDL-AMS modules.

The `+cktpreset` option, with a possible value of `sampld` (`cktpreset=sampld`) enables Spectre or the APS technologies to take fewer time steps during conservative mode simulation while improving simulation resolution at each time step.

Related Topics

- [xrun Command-Line Options for Mixed-Signal Designs](#)
- [Turning On Spectre Multithreading for Device Evaluation](#)
- [Turning On Spectre Parasitic Reduction](#)
- [Loading Plug-In for Spectre Netlist Compiled Functions \(NCFs\)](#)
- [Enabling AMS-APS Mode](#)

Turning on Spectre Parasitic Reduction

If you are using the AMS Designer simulator with the Spectre solver, you can turn on parasitic reduction for designs that have RC parasitics.

To do this:

- Use the `-spectre_args` command-line option to `xrun` to specify the Spectre `+parasitics` command-line argument as follows:

```
xrun -spectre_args +parasitics[= N | rf ] ...
```

Here, the value specified for the `+parasitics` argument represents the maximum frequency (in GHz) of interest for RC reduction. If the chosen value is less than the maximum operating frequency of interest, you may experience accuracy loss for frequencies higher than the specified value.

- `N` represents the user-defined maximum frequency
- `rf` represents the maximum frequency of 30 GHz
- If no value is specified for the `+parasitics` argument, the maximum frequency of 1 GHz is applied by default

Turning on Spectre Multithreading for Device Evaluation

If you are using the AMS Designer simulator with the Spectre solver, you can turn on multithreading for device evaluation.

To turn on Spectre multithreading for AMS simulation:

- Use the `-spectre_args` command-line option to `xrun` (or to `xmsim`) to specify the Spectre `+mt` or `+multithread` command-line argument as follows:

```
xrun -spectre_args +mt[=N] ...
```

```
xrun -spectre_args +multithread[=N] ...
```

Here, `N` is the number of threads. If `N` is larger than the number of cores of the computer, `N` is reset to the number of cores. If `+mt=N` is not specified, the default number of threads is equal to the number of cores of the computer.

Alternatively, you can use the `multithread=on [nthreads=N]` option setting in a control file to turn on multithreading.

Turning off Spectre Multithreading

If you are using the AMS Designer simulator with the Spectre solver, you can explicitly turn off multithreading for AMS simulation.

To do this:

- Specify `-mt` or `-multithread` with the `-spectre_args` command-line option to `xrun` as follows:

```
xrun -spectre_args -mt ...
```

```
xrun -spectre_args -multithread ...
```

Alternatively, you can use the `multithread=off` option setting in a control file to turn off multithreading.

Related Topics

- [xrun Command-Line Options for Mixed-Signal Designs](#)

Simulation Diagnostics

At times, you need additional debugging information to investigate the performance or convergence-related simulation issues. By default, the simulation log file now contains a presimulation summary and a post-transient simulation summary with suggestions on how to speed up the simulation further. In addition, you can use the Spectre command-line option, `+diagnose`, to dump additional debugging information into the simulation log file. The following is an example of using the `+diagnose` option:

```
xrun ... -spectre_args "+aps +diagnose"
```

Here, ... denotes other `xrun` options.

The `+diagnose` option automatically enables the AHDL linter feature, and outputs the following additional information to the log file:

- Convergence-related information for time steps less than 1ps. This can be controlled by using the `warnminstep` option.
- A summary containing statistics about time steps, nodes with convergence issues, and node or device activity information.

The `+diagnose` option has minimal impact on the simulation time, however, it generates a lot of information and may not be appropriate for general use.

When using `spectre_args "+diagnose"` you can also specify the `+transteps` option.

```
spectre_args "+diagnose +transteps"
```

It prints all the transient steps in diagnostic mode.

Related Topics

- [XRUN Invocation for Mixed-Signal Designs](#)

Enabling AMS-APS Mode

To use the APS analog simulator with AMS-Spectre, use the `+aps` argument with the `-spectre_args` option of the `xrun` command.

Example:

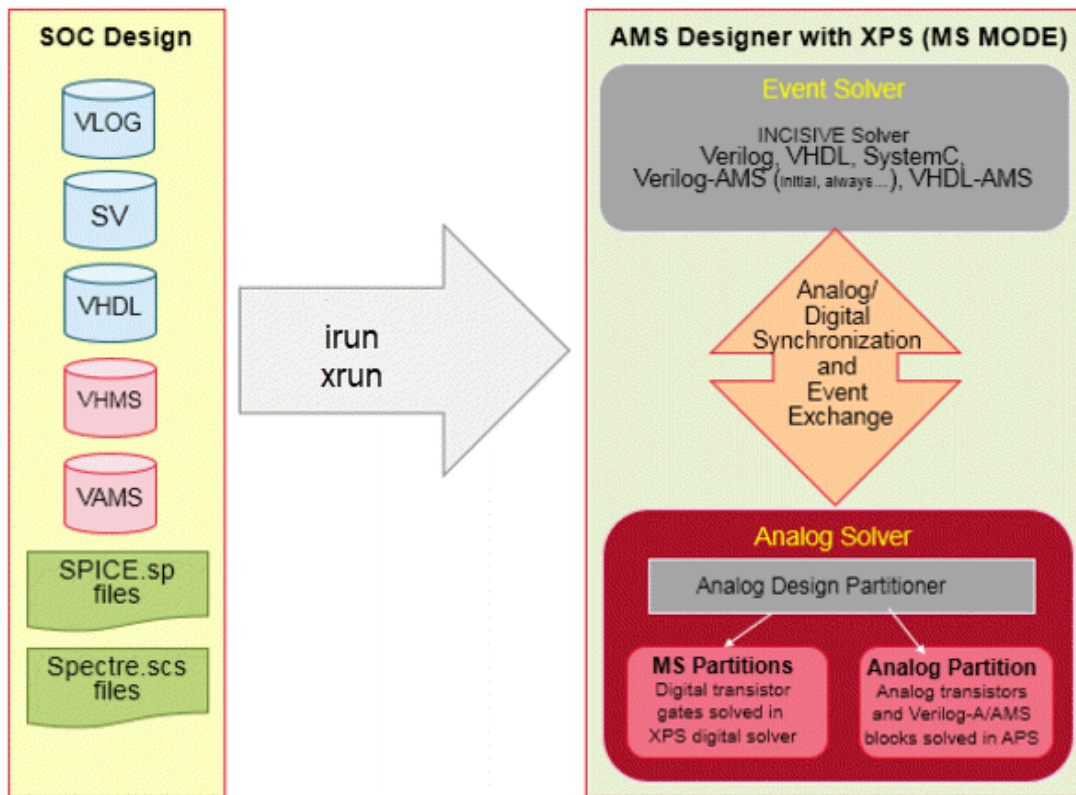
```
xrun -spectre_args "+aps"
```

AMS Spectre XPS MS Mode

Spectre XPS MS delivers a high-performance transistor-level multi-rate simulation solution by combining a highly accurate SPICE engine (Spectre APS) together with a fast digital simulation engine (Spectre XPS). The identification between the analog and digital portions of the circuit is done automatically, with the added flexibility for advanced users to optimize it. The Spectre XPS MS mode is valuable for DUTs including MOS or FinFET device instances in Spectre or SPICE subcircuits.

The AMS Designer simulator supports the Spectre XPS MS simulation mode for AMS applications with mixed-signal transistor level circuits. The AMS Designer simulator first partitions the design between the AMSD event solver and Spectre XPS MS analog solver. Next, the analog block circuit is automatically partitioned into analog and mixed-signal partitions where the digital partition is solved in the XPS solver and the analog partition is solved in APS.

The following figure provides an overview of the AMS Designer simulator with Spectre XPS MS mode:



You should use Spectre XPS MS when there is a large number of gates described by device transistors. For such cases, Spectre XPS MS mode provides additional high-performance and large simulation capacity as compared to Spectre or Spectre APS.

To enable Spectre XPS MS mode in AMS, use `+ms` with the `-spectre_args` option of the `xrun` command.

Example :

```
xrun -spectre_args "+ms" input.scs test.v
```

When the Spectre XPS MS mode is enabled, a message is displayed in the `xrun` log file stating the same. Following is an example of the message that is displayed:

Notice from Spectre.

```
Spectre XPS Mixed -signal Mode Enabled (speed=2)
```

You can specify the errpreset settings with the `+aps` option at the command line to tighten the accuracy, with `+aps=liberal` being the default setting. You can use `+aps=moderate` or `+aps=conservative` setting for more accurate results while compromising on the simulation time.

```
xrun -spectre_args "+ms +aps=moderate" input.scs test.v
```

AMS Spectre XPS MS supports multi-threading computation on multi-core platforms. The multi-threading use model and behavior are identical to Spectre APS.

The AMS Spectre MS mode supports the following features:

- Verilog, SystemVerilog, Verilog-A, Verilog-AMS, VHDL, SystemC
- Transient analysis
- TCL voltage and current probing
- Spectre `save` statement for current and voltage
- SimVision debugger
- Re-invoking the simulation
- AHDL linter
- `+diagnose` option

Adjusting AMS Spectre XPS MS Speed and Accuracy

The Spectre XPS MS mode supports all performance and accuracy settings of Spectre APS for the analog portion. You can use the `++aps` option to improve the performance of the APS analog engine in AMS Spectre XPS MS mode, as shown below.

```
xrun -spectre_args "+ms ++aps" input.scs test.v
```

You can also specify the `errpreset` options with `++aps` to control the speed and accuracy of the analog engine with `liberal` being the default setting which results in faster simulation time but lower analog partition accuracy. Use `++aps=moderate` for moderate analog partition accuracy. However, if accuracy is your primary concern, specify `++aps=conservative` at the command line, as shown below.

```
xrun -spectre_args "+ms ++aps=conservative" input.scs test.v
```

The simulation speed and accuracy of digital portions in the AMS Spectre XPS MS mode can be controlled by the option `+ speed=1/2/3`. The default value is 2, which is sufficient for most applications. Use `+speed=1` for more accurate timing and power simulation of digital portions. Use `+speed=3` for faster digital verification. For example:

```
xrun -spectre_args "+ms ++aps +speed=3" input.scs test.v
```

You can use the `+cktpreset=[sampled|pll]` option to improve the speed of ADC and PLL applications.

The `xrun` log file contains important information on the circuit partitions, simulation time spent on the analog and digital partitions, and suggestions to improve accuracy.

```
*****Pre-Simulation Summary*****
```

```
(MS) Table model. VDD=2.5V used for XPS table model creation, to overwrite tmopt options
```

```
vdd=val
```

```
(MS) Circuit Partitions: 100% transistors and 99.4% nodes are identified as digital
partitions. To force a subcircuit or subcircuit instance to digital, add in the
netlistoptD options cktpreset=digital subckt=[subckt_def_name]optD options
ctkpreset=digital inst=[subckt_inst_name]
```

```
*****Post-Transient Simulation Summary*****
```

```
4.74% simulation time spent in analog partition. Consider force more circuit blocks to
digital, or use more aggressive APS options.95.26% simulation time spent in digital
partition Consider use more aggressive XPS options, add +speed=3 on command line
```

```
*****
```

```
*****Post-
```

Digital Partitioning and Virtual Power Nodes

The Spectre XPS MS mode automatically detects the digital functions in a given circuit. For cases where a design has explicit hierarchical digital or analog functions, you can manually identify the functional blocks (subcircuit or instance) by using the following Spectre options:

```
optD options cktpreset=digital subckt=[dig_ctrl pll]
```

```
optA options cktpreset=analog subckt=[opamp adc]
```

or

```
optD options cktpreset=digital inst=[x1.xdig_ctrl x1.xpll]
```

```
optA options cktpreset=analog inst=[x2.xopamp x2.xadc]
```

You can define the virtual power supply nodes manually by using the `ms_vpn` option that defines the pairs of node names (wildcard is supported) and voltage values. The virtual ground nodes can be defined using the `ms_vgnd` option. The ground voltage is assumed to be 0V.

```
opt1 options ms_vpn = [I1.VDD_DIG 1.8 I1.VDD_DIG1 3.3]
```

```
opt3 options ms_vgnd = [I1.VSS]
```

The xrun log file provides information on the nodes that are identified as power/ground supplies and other nodes that are potential power/ground supplies. A sample log file report is shown below.

Digital Detection Vsource and Virtual Power Supply Node Summary					
	Type	#CC	#Bulk	Voltage	Name

	GND	4274	6866	0	0
	VSOURCE	6386	6866	2.5	vdd
	To force a virtual power or ground node, add in netlist				

In the above report `#CC` refers to the number of MOS channel connected devices, `#BULK` refers to the bulk connections, `Voltage` refers to the node voltage, and `Name` refers to the node name. In the above example, 6386 MOS drain or sources are connected to a vsource on the node named vdd. The node voltage is 2.5 volts.

AMS Spectre XPS MS DC Operating Point Calculation

The Spectre XPS MS mode calculates the DC operating point before performing the transient simulation. By default, the DC operating point for the entire circuit is calculated by Spectre APS. For cases where the Spectre APS operating point calculation takes too long, you can enable the mixed-signal DC calculation - Spectre APS for the analog partition and Spectre XPS for the digital partition, by using the `+msdc=ms` command-line option, as shown below.

```
xrun spectre "+ms +msdc=ms" input.scs test.v //MS DC
```

Multi-Rate IE Partitioning in AMS Spectre XPS MS

All the IEs in AMS are simulated at the same rate (time step size) in the APS block. This process may have a major impact on the simulation speed if a design has high-frequency digital clocks between Verilog or SystemVerilog digital context and SPICE blocks.

The Spectre AMS designer simulator with XPS MS mode has been enhanced such that if an IE is connected to an XPS block (also called MS partition), it is computed with the same time step as the MS partition. When there are multiple IEs connected to different MS partitions (or XPS blocks), they are simulated at a different rate. As a result, the time step of an APS block is not limited by such IEs and the simulation can run much faster.

AMS IEs that connect only to the XPS block are tagged as XPS-IEs and are partitioned in XPS block also. AMS `inh_conn` IEs that have Ain/Aout port connected to an XPS block and power supply port (`in` port) connected to APS block are also partitioned in XPS. All other AMS IEs are tagged as APS-

IE. Verilog-AMS or VHDL-AMS models with no IE remain in APS block.

Related Topic

- [xrun Command-Line Options for Mixed-Signal Designs](#)

AMS Designer with Spectre FX

Spectre FX is a next-generation FastSPICE simulator that addresses simulation challenges such as the increasing demand for performance, accuracy, and capacity from transistor-level simulators. It also addresses the requirements of ever-shrinking process nodes, the adoption of new design practices, the need for full-chip simulation, and the increasing influence of parasitic elements, which greatly increase the size and complexity of the designs to be simulated.

Unlike the other Spectre simulators, such as Spectre APS, Spectre X, and Spectre XPS, the default compatible mode in Spectre FX is SPICE, which means that you no longer need to add the `+spice` argument in the command line for the SPICE syntax. However, you do need to add the `+spectre` argument in the command line for the Spectre syntax.

Spectre FX supports only transient analysis and uses the Unified Database (UDB) format by default.

To use the Spectre FX analog simulator with Spectre AMS Designer:

- Use the `+fx` argument with the `-spectre_args` option of the `xrun` command. The default simulator used is SPICE. To use the Spectre simulator, specify the `+spectre` option.

Example

```
xrun -spectre_args "+fx +spectre"
```

To change the solver, use the `+preset` option.

```
xrun -spectre_args "+fx +spectre +preset=[lx | ax | mx | lx | vx | baseax |  
basemx | baselx | basevx]"
```

The simulation tolerance for voltage and current calculation can be done by setting the `+speed` option to `lx`.

```
xrun -spectre_arg "+fx +spectre +preset=vx +speed=lx +mt=8"
```

Spectre FX supports multithreading on multi-core computer platforms. The default behavior is to use a single thread.

Example

The following command enables multithreading on eight cores.

```
-spectre_arg "+fx +spectre +mt=8"
```

Options for AMS Designer with Spectre FX

Spectre FX supports both Spectre and SPICE syntax to set the options.

- Spectre syntax:

```
SimOpt1 options option_name1=option_value option_name2=option_value ...
```

- SPICE syntax:

```
.option option_name1=option_value option_name2=option_value ...
```

Connect Module Usage

You can enable UCM for AMS Designer with Spectre FX using the command `accurate=moderate`, where `moderate` is the default value.

The following table shows the default value changes for the default connect modules, compared to other analog simulators.

	Analog Solver		
	Spectre, APS, SpectreX, MS	SpectreFX	
	Default: legacy CR_full_fast	Default: UCM accuracy=moderate	
tr	200p	10p	Rise or full time of transition between vhi and vlo

With Spectre FX, the default IE card parameter value `tr=10p` lets you convert logic clocks to electrical clocks with up to 50 Giga Hertz frequency. With the parameter value `tr=200p`, the clock frequency is limited to 2.5 Giga Hertz.

Related Topic

[Starting a Spectre FX Simulation Run](#)

Loading Plug-In for Spectre Netlist Compiled Functions (NCFs)

To load the plug-in for netlist compiled functions (NCFs), use the `-spectre_args` command-line option to pass the `-plugin` command-line option to Spectre. For example:

```
xrun -spectre_args "-plugin libmyplugin_sh.so"
```

Alternatively, you can use the `loadplugin` command in your Spectre netlist file. For example:

```
loadplugin "libmyplugin_sh.so"
```

See "Netlist Compiled Functions (NCFs)" in the Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide for information about NCFs.

AMS Designer Simulator for Design Verification

You can use the AMS Designer simulator for mixed-signal simulation in your system-on-chip (SoC) design verification process. You can run `xrun` to simulate designs containing digital and analog design units. `xrun` supports hardware description languages such as Verilog and Verilog-AMS, and analog netlist formats such as SPICE and Spectre. You can incorporate SPICE blocks into otherwise Verilog-centric simulations. The program sends analog portions directly to the analog engine for parsing and elaboration. You can perform checks and measurements on design considerations such as dynamic power consumption and current leakage.

For better yield ratio during chip design, you might want to replace some Verilog modules with corresponding SPICE or Spectre subcircuits to achieve device-accurate simulation results during design verification. You can use a port-bind file to specify how you want the software to map port names (scalars and buses) when you replace a Verilog module with a SPICE or Spectre equivalent. You can also reference the original Verilog file so that the software can map the SPICE or Spectre ports by order.

Related Topics

- [Creating a Testbench](#)
- [XRUN Invocation for Mixed-Signal Designs](#)
- [Port Bindings between Verilog and SPICE](#)
- [Customized Port-Bind File](#)

Creating a Testbench

The major tasks for creating a testbench are as follows:

1. Create a top level for your design by connecting and instantiating the digital and analog components.
In the following excerpt, wires `vcoclk`, `clock_2`, `clock_1`, `clock_0`, `net036`, and `p0` connect to the digital instances (`counter` and `divider`) and the analog instance (`pll_top`):

```
// Testbench`
timescale 1ps/1ps
module testbench ();

...
wire vcoclk, clock_2, clock_1, clock_0, net036, p0;

...
counter counter (reset, vcoclk, clock_2, clock_1, clock_0);
divider divider (vcoclk, net036, p0, reset);
pll_top pll_top (refclk, reset, vcoclk, clock_2, clock_1, clock_0, net036, p0,
clk_p0_1x, clk_p0_4x);

endmodule
```

In a testbench file, you do not have to declare the `electrical` discipline for any wires even when you use them to connect to the ports of analog instances. You therefore do not need to include the `disciplines.vams` file.

2. Create the stimuli to the device under test (DUT).

In the following excerpt, we add reset and `refclk`:

```
// Testbench`
timescale 1ps/1ps

module testbench ();
  reg reset; reg refclk;
  ...
  wire vcoclk, clock_2, clock_1, clock_0, net036, p0;

  initial begin
    reset=1;
    #200 reset=0;
  end

  initial begin
    refclk=0;
    #200 refclk=1;
  end
  always #2500 refclk=~refclk;
  ...
  counter counter (reset, vcoclk, clock_2, clock_1, clock_0);
  divider divider (vcoclk, net036, p0, reset);
  pll_top pll_top (refclk, reset, vcoclk, clock_2, clock_1, clock_0, net036, p0,
    clk_p0_1x, clk_p0_4x);

endmodule
```

3. (Optional) Monitor or self-check the output.

You can monitor the output by adding `$monitor` to your module definition:

```
initial begin
    $monitor (y);
end
```

Related Topic

- [Design Configuration Using Mixed Languages](#)

Run Script for xrun

When creating an `xrun` run script for design verification, you need to specify the following items on the `xrun` command line:

Item/Option	Example	
Digital design inputs	<code>xrun</code>	<code>./source/digital/*.v \</code>
AMS control file		<code>amsdcb.scs \</code>
Analog solver control file		<code>top.scs \</code>
Timescale for undefined Verilog modules		<code>-timescale 1ns/100ps \</code>
Tcl probe on behavioral nodes		<code>-input probe.tcl</code>

Some things to note:

- You can specify the AMS and analog control file on the command line, just like any other input file.
- You do not need to specify any connect rules (`-amsconrules`) when you have an `ie` statement in an `amsd` block.

If you do use a connect rules file, you can specify it as a regular input file, directly on the command line.

- The model files must be included in the AMS control file.

For example:

```
include "./models/resistor.scs" section=res
include "./models/diode.scs" section=dio
include "./models/pmos1.scs" section=nom
include "./models/nmos1.scs" section=nom
```

Tcl File Creation to Probe Behavioral Nodes

To probe behavioral nodes and save that information to a database file called `waves.shm`, you might create a Tcl file (`probe.tcl`) containing commands such as the following:

```
database -open waves -into waves.shm -defaultprobe -create -database waves -all -depth
allprobe -create -database waves testbench.refclkprobe -create -database waves
testbench.clk_p0_1xprobe -create -database waves testbench.clk_p0_4xprobe -create -
```

```
database waves testbench.p0
```

You can also specify simulation control commands (such as `run`) in a Tcl file.

Related Topics

- [amsconnrules](#)
- [amsd Block](#)
- [run](#)
- [xrun Command Syntax](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)

Port Bindings between Verilog and SPICE

Port-binding considerations include case mapping and bus connections between Verilog and SPICE. The following guidelines can help you choose which method to use for your design requirements:

Design Requirements	Port-Binding Method
All buses have uniform ascending/descending port order and all ports have uniform case mapping	autobus
Customized port binding requirements	port-bind file
You have a Verilog version of a SPICE block	portmap reffile
You need to bind ports by name, rather than by order	portmap porttype

Binding Ports using autobus

To bind Verilog to SPICE ports using `autobus`, do the following:

1. Use an include statement to include the SPICE design file:

```
include "analog_top.sp"
```
2. Use `portmap` and `config` cards in the `amsd` block to specify the SPICE file and subcircuits

that you have instantiated in Verilog modules:

```
amsd{
    portmap subckt=analog_top autobus=yes busdelim="<>"
    config cell=analog_top use=spice
}
```

The default value for `autobus` on the `portmap` statement is `yes`.

3. Specify the AMS control file directly on the `xrun` command line:

```
xrun ... amsdcb.scs
```

Here is an example. The top module, `testbench`, instantiates one SPICE block, `pll_top`:

```
module testbench ();
    reg myRESET;
    reg refclk;
    wire [1:0] clk_out;
    .....
    pll_top p1(refclk, myRESET, clk_out);
endmodule

//The SPICE subcircuit definition looks like this:

.subckt pll_top refclk reset P0_CLK[1] P0_CLK[0]
...
.ends pll_top

//You specify autobus port binding using the SPICE pll_top subcircuit in the amsd
block as follows:

amsd{
    portmap subckt=pll_top autobus=yes
    config cell=pll_top use=spice
    ...
}

//The elaborator generates the following port-bind file:

//
//portmap file for spice subckt pll_top
refclk :      refclk  dir=inout
reset  :      reset   dir=inout
{ P0_CLK[1], P0_CLK[0] } :      P0_CLK[1:0]      dir=inout
```

The `SPICEname` identifiers (on the left side of the colon) come from the SPICE subcircuit definition (such as `.subckt pll_top refclk reset P0_CLK[1] P0_CLK[0]`).

The `VerilogName` identifiers (on the right side of the colon) represent the port names of the Verilog module that the software generates internally. These port names can be different from the actual interface signal names in the instantiation in the top-level testbench module. In this example, for instance, the interface signal names are `refclk`, `myRESET`, and `clk_out`, while the port names in the port-bind file are `refclk`, `reset`, `P0_CLK[1]`, and `P0_CLK[0]`.

You should change the port directions to match your design requirements (`dir=inout` is the default direction setting for SPICE ports).

The software maps `P0_CLK[1]` and `P0_CLK[0]` into a bus because `[]` is the default bus delimiter.

Binding Ports using a Port-Bind File

Some designs require customized port binding, such as mixed-case mapping, more complicated bus forms such as those that include concatenations, and mixed ascending and descending bus orders. You can use a port-bind file to specify your customized port binding requirements.

You can use autobus to generate the initial port-bind file (`.pb`) automatically, then you can customize the mappings in that file. The general steps you can follow are:

1. Use autobus to generate the initial port-bind file (`.pb`).
2. Customize port mappings in the initial port-bind file.
3. Use `portmap ... file=` to specify the port-bind file you customized.

```
include "analog_top.sp"

amsd{

    portmap subckt=analog_top file="analog_top.pb"
    config cell=analog_top use=spice
}
```

Binding Ports using a Verilog File

If your design contains units for which you have both a Verilog and a SPICE version, you can use a Verilog file to specify your binding options. Perhaps you simulate a purely digital version of your design first, and then selectively replace some blocks with a SPICE version.

You can specify a file containing a Verilog module that defines the port mappings to use from a Verilog parent to a SPICE subcircuit instance using `portmap reffile` in your AMS control file. This

approach lets you replace the interface of a subcircuit with that of a Verilog module.

For example:

```
include "analog_top.sp"
amsd{
    portmap subckt=analog_top reffile="analog_top.v" refformat=verilog
    config cell=analog_top use=spice
}
```

In this example, `analog_top.v` is the Verilog file that contains the port binding information your design requires. The software applies port bindings (interfaces) defined in `analog_top.v` to instances of `analog_top`. While the elaborator uses the port bindings you define for `analog_top` in `analog_top.v` to determine how to connect instances of `analog_top`, the simulator simulates the `analog_top` subcircuit you define in `analog_top.sp`.

For example, `analog_top.v` might contain the following:


```
module analog_top (in1, itune, in2);
  inout in1;
  inout [0:1] itune;
  inout in2;

  analog beginend
end
module

//The analog_top subcircuit in analog_top.sp might look like this:

.subckt analog_top
+ in1 itune[0] itune[1] in2

...
.ends analog_top

//When you instantiate a subcircuit called analog_top in module top like this:

module top (ext_clk, pll_clk);
  input ext_clk, pll_clk;

  wire [0:1] itune;
  wire res;

  analog_top xana_top(
    .in2(pll_clk),
    .itune(itune),
    .in1(ext_clk)
  );

  ...
endmodule
```

Binding ports by order is not recommended when the ports in the Verilog reference file and the ports in the equivalent SPICE subcircuit do not match by order. In such situations, it is recommended that you bind ports by name using the `porttype=name` parameter.

Binding Ports by Name

When you need to bind the ports in your reference file by name, rather than by order, you can use the `porttype` specifier in a `portmap` statement. For example:

```
amsd{  
    ie vsup=1.8  
    portmap module=counter reffile="./source/digital/counter.v" porttype=name  
    config cell=counter use=hdl  
}
```

The software uses this specifier when automatically generating a port-bind file. The default binding mechanism is by-order (`porttype=order`). However, if your design consists mostly of SPICE and Verilog ports that have the same name, but with mismatched port order, you might get a more accurate port-bind file by specifying `porttype=name`.

If the software cannot match ports between Verilog and SPICE successfully, the elaborator writes " not_found " as a placeholder in the port-bind file it generates (in runDir /portmap_files). You can edit this file to change all not_found keywords to the correct Verilog port names, then use the `file` parameter (instead of `reffile` and `porttype`) in a `porttype` statement to specify the file you saved, and run the simulation again.

Related Topics

- [autobus](#)
- [portmap](#)
- [porttype](#)
- [config](#)
- [file](#)
- [reffile](#)
- [XRUN Invocation for Mixed-Signal Designs](#)
- [Creating a Customized Port-Bind File](#)
- [Customized Port-Bind Files](#)

Customized Port-Bind Files

A port-bind file contains information about how you want the SPICE subcircuit ports mapped to Verilog buses. You can specify customized bus element mappings as well as port direction. The general format for port bindings is as follows:

```
SPICEName : VerilogName [ dir=input|output|inout]
```

where `SPICEName` and `VerilogName` are identifiers that do not have to match.

Each `SPICEName` corresponds to a node name or bus in the SPICE subcircuit definition.

Each `VerilogName` corresponds to a wire name or bus in the Verilog module. The port direction specifier is optional.

For scalar nodes, the format is as follows:

```
node1 : NODE1
```

For a vector, the format is as follows:

```
{ myBus_0, myBus_1 } : myBUS[0:1]
```

You can specify a range of bus elements as follows:

```
{ busA[0]-busA[10] } : BusA[0:10]
```

You can specify a customized mapping of elements as follows:

```
{ busA[10]-busA[5], busA[0]-busA[4] } : busA[0:10]
```

You can specify a customized mapping of random elements as follows:

```
{ a_0, a_1, a_2, a_4, b, abc } : bus[0:5]
```

You might have specified a unary bus delimiter such as `&` or `#` :

```
{ uBus&0, uBus&1, uBus&2, vBus#3, vBus#4, vBus#5 } : bus[0:5]
```

The following default rules apply to any node or bus that you do not explicitly specify in the port-bind file:

- Port names match exactly (name-to-name), including casing
- Bus delimiters are `[]` and `<>`

These default rules are the same as those that apply when you use `autobus`.

Customized Port-Bind File Examples

Here is an example that shows how you can use a port-bind file to define port binding between Verilog and SPICE blocks in a very general way. Study the port-bind file format carefully to understand how you can create custom port bindings according to your connection requirements.

You might instantiate subcircuit `analog_top` in module `top` as follows:

```
module top (ext_clk, pll_clk);
input ext_clk, pll_clk;

wire [0:1] itune;
wire res;

analog_top xana_top(.in2(pll_clk),
.itune(itune),
.in1(ext_clk));

...
endmodule
```

Subcircuit `analog_top` (in `analog_top.sp`) might look like this:

```
.subckt analog_top
p+ in1 itune[0] itune[1] in2
...
.ends analog_top
```

Your `amsd` block might contain the following:

```
include "analog_top.sp"
amsd{
    portmap subckt=analog_top file="analog_top.pb"
    config cell=analog_top use=spice
}
```

If your port-bind file, `analog_top.pb`, contains the following:

```
in1 : in2          dir=inout
{ itune[1], itune[0] } : itune[0:1]  dir=inout
in2 : in1          dir=inout
```

the elaborator derives the following information from these port bindings:

1. Port `in1` of SPICE subcircuit `analog_top` connects to the net `in` in module `top` that connects to formal port `in2` in instance `xana_top` of `analog_top`.
The elaborator connects net `pll_clk` to port `in1` of SPICE subcircuit `analog_top`.

2. Port `itune[1]` of SPICE subcircuit `analog_top` connects to the net in module `top` that connects to formal port `itune[0]` in the instance `xana_top` of `analog_top`.
The elaborator connects net `itune[0]` to port `itune[1]` of SPICE subcircuit `analog_top` and follows the same logic to connect net `itune[1]` to port `itune[0]` of SPICE subcircuit `analog_top`.
3. Port `in2` of SPICE subcircuit `analog_top` connects to the net in the module `top` that connects to formal port `in1` in instance `xana_top` of `analog_top`.
The elaborator connects net `ext_clk` to port `in2` of SPICE subcircuit `analog_top`.

Here is another example:

```
// mrcg.v -- Verilog top
module mrcg (ext_clk, pll_clk);
ANALOG_TOP xana_top(.Q_1(Q_1), .Q_2(Q_2), .IN2(pll_clk), .ITUNE(ITUNE),
.IN1(ext_clk));
endmodule

// analog_top.cir -- spice leaf
.subckt analog_top q_1 q_2 IN1 itune_0 itune_1 in2
.ends analog_top

// analog_top.pb -- port-bind file
          q_1 : Q_1           dir=input
          q_2 : Q_2           dir=input
          IN1 : in1           dir=inout
{ itune[1], itune[0] } : ITUNE[0:1] dir=input
          in2 : IN2           dir=inout

// amsd.scs
include "analog_top.cir"
amsd{
    portmap subckt=analog_top file="analog_top.pb"
    config cell=ANALOG_TOP use=spice
}
```

Rules That Apply to Customized Port-Bind Files

Here are some rules to remember when using port-bind files:

- The location where the elaborator generates the port-bind files is `runDir/portmap_files` where `runDir` is the directory where you run `xrun`.

- If a port-bind file for a certain subcircuit already exists in the default location (mentioned in the previous bullet), the elaborator will not overwrite the file and it will not generate a port-bind file for this case.
- The elaborator will never use a port-bind file unless you specify it explicitly in the AMS control file using `portmap ... file`. The software will issue a message that clearly indicates whether the elaborator used a port-bind file.
- The port-bind file you specify (using `portmap ... file`) must have a valid UNIX path, either absolute or relative to the directory where you run `xrun`.

Any mappings you specify in a port-bind file or in a Verilog module that defines port mappings take precedence over any options you specify explicitly using other options such as `autobus`, `casemap`, and `busdelim`.

Related Topics

- [autobus](#)
- [casemap](#)
- [busdelim](#)
- [Port Bindings between Verilog and SPICE](#)

Multiple Power Supplies for Designs

You can use the following methods to use multiple power supplies for designs

- Using the `ie` Statement in an `amsd` Block for Multiple Power Supply Design: Lets you specify a supply value to apply to a library, cell, or instance in your design. You can also customize Cadence-provided connect rules. This method requires no knowledge of Verilog-AMS disciplines.
- Using Block-Based Discipline Resolution for Multiple Power Supply Design: Lets you use the `-setdiscipline` option to tell the elaborator which discipline to apply to which design scopes.
- Creating Supply-Sensitive Modules for Multiple Power Supply Designs: Lets you manage clean and accurate analog-to-digital and digital-to-analog connections in a mixed-signal design that has multiple power supplies and multiple voltages using supply-sensitive connect modules (CMs).
- Inserting Interface Elements Selectively for Multiple Power Supply Designs: Lets you use the `-ams_check_msup` option to enable selective insertion of individual interface elements (IEs).

Related Topics

- [ie Statement in an amsd Block for Multiple Power Supply Design](#)
- [Supply-Sensitive Modules for Multiple Power Supply Designs](#)
- [Block-Based Discipline Resolution for Multiple Power Supply Design](#)
- [Interface Elements for Multiple Power Supply Designs](#)

ie Statement in an amsd Block for Multiple Power Supply Design

If you are moving a purely-digital or purely-analog design into the analog/mixed-signal (AMS) domain for full-chip verification, you can use the `ie` statement in an `amsd` block to set up connect rules for multiple power supply design.

When you use the `ie` statement, you do not need to use `-amsconnrules` to specify your connect rules. You also do not need to specify a connect module path or to compile any connect modules. You do not need to use the `-discipline` option.

Supply Value to Apply to a Library, Cell, or Instance

Use the `ie` statement as follows to specify a supply value to apply to a library, cell, or instance:

```
amsd{
  ...
  ie vsup= supplyValue [ library_cell_or_instance ]
  ...
}
```

The simulator uses the `supplyValue` as the final real value for logical `1` and applies it to the library, cell, or instance you specify. The software automatically builds the "full-fast" connect rule with the voltage supply level you specify. For example, if you specify

```
amsd{
  ...
  ie vsup=1.8
  ...
}
```

the software automatically builds the "full-fast" connect rule for a 1.8-Volt supply from the Cadence-provided full-fast connect rule.

To specify a supply level for a particular library, cell, or instance, use one of the scope assignment parameters. For example, to specify a 1.8-Volt supply globally, and 5.0 Volts and 3.3 Volts to particular libraries, you might use the following `ie` statements:

```
amsd{
  ie vsup=1.8
  ie vsup=5.0 lib="50v_lib"
  ie vsup=3.3 lib="33v_lib"
}
```


To specify a 1.8-Volt supply globally, and a 3.3-Volt supply for a particular cell, you might use the following `ie` statements:

```
amsd{
    ie vsup=1.8
    ie vsup=3.3 cell="vlog_buf_1channel"
}
```

Finally, to specify a 1.8-Volt supply globally, and a 3.3-Volt supply for particular instances, you might use the following `ie` statements:

```
amsd{
    ie vsup=1.8
    ie vsup=3.3 inst="testbench.vlog_buf_channel1"
    ie vsup=3.3 inst="testbench.vlog_buf_channel2"
}
```

You must specify the full hierarchical path to the instances.

The software writes information about these connect rules to the `xrun.log` file.

Cadence-Installed Connect Rules Customization

You can customize the Cadence-provided full-fast connect rule by specifying other parameters on the `ie` statement, such as `vthi`, `vtlo`, and `tr`. For example, the following `ie` statement specifies a 2.0-Volt full-fast connect rule with a customized rise time for the analog transition (0.1 ns):

```
amsd{
    ...
    ie vsup=2.0 tr=0.1n
    ...
}
```

You can customize other Cadence-provided connect rules by specifying the `connrules` parameter assignment in the `ie` statement. For example:

```
ie connrules=full vsup=2.0 tr=0.1n
```

The set of Cadence-provided connect rules does not include a rule for a 2.0-volt supply and a 0.1 ns transition time. However, using the `ie` statement above, you can specify that you want the software to customize the Cadence provided "full" connect rule with these parameters.

Cadence-Provided Connect Rules

See `your_install_dir/tools/affirma_ams/etc/connect_lib` for the set of connect rules files that Cadence provides. See `your_install_dir /tools/affirma_ams/etc/connect_lib/README` for detailed information about them.

Related Topics

- [ie](#)
- [Scope Assignments](#)
- [Block-Based Discipline Resolution for Multiple Power Supply Design](#)
- [Interface Elements for Multiple Power Supply Designs](#)
- [Supply-Sensitive Modules for Multiple Power Supply Designs](#)

Block-Based Discipline Resolution for Multiple Power Supply Design

The Spectre®AMS Designer simulator complies strictly with the Verilog®-AMS discipline resolution process, reliably identifies interdomain connectivity, and inserts connect modules automatically. Using block-based discipline resolution (BDR), you can use your knowledge of the analog and digital blocks in your design to control the search space for the discipline resolution process, thus improving elaboration performance. You can also use BDR to set different disciplines in the digital domain for designs that have more than one power supply.

Connect Modules, Connect Rules, and Discipline Definitions

Cadence provides a set of connect modules ([CMs](#)) and connect rules in the IUS installation hierarchy:

`your_install_dir/tools/affirma_ams/etc/connect_lib`

You can create a custom file that contains connect rules, connect module parameters, and discipline definitions to suit your multiple power supply design requirements.

In AMS Designer, the default digital discipline is `logic` and the default analog discipline is `electrical`. You can create custom disciplines that contain information about the power

supplies in your design. The elaborator uses BDR to determine which part of the design has which power supply.

To create custom connect rules, connect modules, and discipline definitions, do the following:

1. Copy the template connect rules file (`ConnRules*.vams`) from the `connect_lib` directory in the installation hierarchy to your local area.
2. Save the file under another name, such as `ConnRules_multpower.vams`.
3. Customize the connect module parameters (such as `Vsup`, `Vthi`, and so on) according to the what is required for your design.
For example, you might define three different sets of `Vsup`, `Vthi`, and `Vtlo` for 1.0V, 1.8V, and 3.3V power supplies:

```
-----  
// Parameter Customization  
`define Vsup1 1.0  
`define Vsup2 1.8  
`define Vsup3 3.3  
`define Vthi1 0.66  
`define Vthi2 1.2  
`define Vthi3 2.2  
`define Vtlo1 0.33  
`define Vtlo2 0.6  
`define Vtlo3 1.1  
-----
```

4. Define disciplines to represent the different power supplies in your design.
For example, you can define disciplines `logic18V` and `logic33V` to represent 1.8V and 3.3V power supplies as follows:

```
-----  
// Definition of Disciplines  
discipline logic33V  
    domain discrete;  
enddiscipline  
  
discipline logic18V  
    domain discrete;  
enddiscipline  
-----
```

The default discipline is `logic` , which is also of `discrete` domain.

Create customized connect rules using your custom disciplines.

For example, the following connect rules use the `logic18V` discipline:

```

-----
// ConnRules Specification
connectrules ConnRules_mulptower;
  connect L2E #(
    .vsup(`Vsup2), .vthi(`Vthi2), .vtlo(`Vtlo2),
    .tr(`Tr), .tf(`Tr), .tx(`Tr), .tz(`Tr),
    .rlo(`Rlo), .rhi(`Rhi), .rx(`Rx), .rz(`Rz) )
  input logic18V, output electrical;
  connect E2L #(
    .vsup(`Vsup2), .vthi(`Vthi2), .vtlo(`Vtlo2), .tr(`Tr) )
  input electrical, output logic18V;
  connect Bidir #(
    .vsup(`Vsup2), .vthi(`Vthi2), .vtlo(`Vtlo2),
    .tr(`Tr), .tf(`Tr), .tx(`Tr), .tz(`Tr),
    .rlo(`Rlo), .rhi(`Rhi), .rx(`Rx), .rz(`Rz) )
  inout electrical, inout logic18V;
endconnectrules
-----

```

Once `ConnRules_mulptower.vams` contains all the custom parameters, disciplines, and connect rules, the elaborator can use this information during discipline resolution to detect the discipline pairs (such as `logic18V` and `electrical`) and insert the proper connect module with the proper power supply.

The timescale value has changed from 1ns/100ps to 1ns/1ps in all E2L*, L2E*, Bidir* connect modules contained in the `connectLib` library of the AMS Designer tool installation. This provides accurate simulation time resolution for any system with clock rise/fall time of 125 femtosecond.

The connect modules `timescale` precision should be maintained to verify this logic expression (`tr> timescale_precision/8` and `tf> timescale_precision/8`). In Cadence building CMs, the default is `time_precision = 1ps`.

If the system to be simulated needs a clock rise/fall time less than 125 fs, specify the timescale with `-OVERRIDE_Precision` option in the `xmelab/xrun` command. For example:

```
xrun -timescale 1ns/100fs -OVERRIDE_Precision
```

Custom Connect Modules, Connect Rules, and Disciplines on the Command Line

You can specify your custom connect modules and connect rules on the `xrun` command line using the `-amsconrules` option. For example:

```
xrun ConnRules_mulptower.vams -amsconrules ConnRules_mulptower ...
```

You can use the `-setdiscipline` option to tell the elaborator which discipline to apply to which design scopes:

```
xrun ... -setdiscipline " disciplineSpecification "...
```

Suppose your design contains a block called `vlog_buf` with one channel that has a 1.8V power supply and another that has a 3.3V power supply. You can set the discipline on the individual terminals instead of on a cell or instance using the `instterm` specifier. For example, the following options specify the `logic33V` discipline for instance terminals `testbench.vlog_buf.I1.in`, `testbench.vlog_buf.in_33v`, `testbench.vlog_buf.out_33v`, and `testbench.vlog_buf.I2.out` :

```
-setd "instterm-testbench.vlog_buf.I1.in- logic33V" \  
-setd "instterm-testbench.vlog_buf.in_33v- logic33V" \  
-setd "instterm-testbench.vlog_buf.out_33v- logic33V" \  
-setd "instterm-testbench.vlog_buf.I2.out- logic33V"
```

Other terminals in `vlog_buf` retain the default discipline, which you can specify using the `-discipline` option. For example:

```
xrun ... -discipline logic18V ...
```

Related Topics

- [Discipline Resolution Methods](#)
- [Supply-Sensitive Modules for Multiple Power Supply Designs](#)
- [XRUN Invocation for Mixed-Signal Designs](#)

Supply-Sensitive Modules for Multiple Power Supply Designs

You can manage clean and accurate analog-to-digital and digital-to-analog connections in a mixed-signal design that has multiple power supplies and multiple voltages using supply-sensitive connect modules (CM s). CMs convert signal values from one domain (analog/electrical) to the other (digital/logic/discrete). The software automatically inserts CMs between digital ports and analog nets.

Analog-to-digital conversions typically involve comparing an analog signal to a predetermined threshold voltage: Analog signal values above the threshold become a logic `1`. Analog signal values below the threshold become a logic `0`.

Digital-to-analog conversions typically involve mapping a logic- 0 -to-logic- 1 transition to a ramp from a predetermined analog voltage value that corresponds to a logic 0 to a higher analog voltage value that corresponds to a logic 1 , and a logic- 1 -to-logic- 0 transition to a ramp from a predetermined analog voltage value that corresponds to a logic 1 to a lower analog voltage value that corresponds to a logic 0 .

A connect module that is aware of the power supply to which a digital port will connect in the final physically-realized transistor design can use the power supply information to determine which analog voltages correspond to logic high (1) or logic low (0) values, and what the high and low voltage threshold values are for analog-to-digital conversions. Using supply-aware--or supply- sensitive --CMs, you can model the interfaces between analog and digital domains regardless of which power supplies apply to the various digital blocks in the design. You use `supplySensitivity` and `groundSensitivity` attributes to specify the sensitivity. When you use these sensitivity attributes, you make a connect module sensitive to the signals on a digital port, regardless of the port direction.

To set up your multiple power supply design to use supply-sensitivity CMs, do the following:

1. Specify the necessary attributes in the connect module.
2. Add the corresponding attributes to the connected digital port definition in the ordinary module.

If the connected digital port is part of a schematic, you define the attributes on the connected pin on the schematic. See "Using Net and Pin Properties" in the Virtuoso® AMS Designer Environment User Guide for more information.

3. (Optional) Use detailed discipline resolution:

```
xrun ... -disres detailed ...
```

Supply-Sensitive Connect Modules

To create supply-sensitive analog-to-digital and digital-to-analog connect modules (CM s), add supply-sensitivity attributes to the power and ground pin declarations. Typically, the supply nets you specify using these attributes are global signals.

Here is an example of a supply-sensitive analog-to-digital connect module:

```
// Supply-sensitive analog-to-digital
connect module connectmodule elect2logic(aVal, dVal);

    input aVal;
    output dVal;
    electrical aVal;
    logic dVal;

    electrical (* integer supplySensitivity = "cds_globals.\\vdd! " ; *) VDD ;
    electrical (* integer groundSensitivity = "cds_globals.\\vss! " ; *) VSS ;

    reg temp;
    real vth_hi, vth_lo, vdd, vss, swing; // threshold and rail voltages

    assign dVal = temp // reg temp drives the output dVal

    analog begin
        @ ( initial_step ) begin
            // calculate rail voltages and voltage swing
            vdd = V(VDD);
            vss = V(VSS);
            swing = vdd - vss;
            // calculate threshold voltages
            vth_hi = vss + (swing * 0.6);
            vth_lo = vss + (swing * 0.4);
        end // initial_step
    end // analog block

    // whenever analog value rises above high threshold, digital value becomes 1
    always @ ( above(V(aVal) - vth_hi) ) temp = 1'b1;

    // whenever analog value falls below low threshold, digital value becomes 0
    always @ ( above(vth_lo - V(aVal)) ) temp = 1'b0;

endmodule
```

Here is an example of a supply-sensitive digital-to-analog connect module:

```
// Supply-sensitive digital-to-analog

connect module connectmodule logic2elect(dVal, aVal);
```

```

inout dVal;
output aVal;
logic dVal;
electrical aVal;

parameter rout=50; // output impedance, 50 Ohms
parameter real td = 1n, tr = 1n, tf = 1n;
reg temp;

electrical n; // an intermediate node
real vth_hi, vth_lo, vdd, vss, swing; // threshold and rail voltages

electrical (* integer supplySensitivity = "cds_globals.\\vdd! "; *) VDD ;
electrical (* integer groundSensitivity = "cds_globals.\\vss! "; *) VSS ;

analog begin
    @ ( initial_step ) begin
        // calculate rail voltages and voltage swing
        vdd = V(VDD);
        vss = V(VSS);
        swing = vdd - vss;
        // calculate threshold voltages
        vth_hi = vss + (swing * 0.6);
        vth_lo = vss + (swing * 0.4);
    end // initial_step

    V(n) <+ transition( (dVal ==1 ? vdd : vss), td, tr, tf );
    I(a,n) <+ V(n) / rout; // models output impedance
end // analog block

assign dVal = temp // bind dVal to register temp

// propagate digital value 1 to receivers when rise above vth_hi
always @ ( cross (V(aVal) - vth_hi, +1) ) temp = 1'b1;

// propagate digital value 0 to receivers when fall below vth_lo
always @ ( cross (V(aVal) - vth_lo, -1) ) temp = 1'b0;

endmodule

```


Supply-Sensitivity Attributes to an Ordinary Module

In an ordinary module, add supply-sensitivity attributes to the input and output pin declarations, naming the supplies to which they are sensitive. Here is an example that illustrates how you would add `supplySensitivity` and `groundSensitivity` attributes to the input and output pins of a simple digital inverter module to specify their sensitivity to `pwr1` and `gnd1` supply pins.

```
module inv (a, y, pwr1, gnd1);

    inout pwr1, gnd1;

    input (* integer supplySensitivity = "pwr1";
          integer groundSensitivity = "gnd1"; *) a;

    output (* integer supplySensitivity = "pwr1";
            integer groundSensitivity = "gnd1"; *) y;

    not il (.y(y), .a(a));

endmodule
```

Note that you can also declare your power and ground nodes using inherited connections, such as

```
module inv (a, y, pwr1, gnd1);

    inout (* integer inh_conn_prop_name = "vdd";
           integer inh_conn_def_value = "cds_globals.\\vdd! "; *) pwr1;

    inout (* integer inh_conn_prop_name = "vss";
           integer inh_conn_def_value = "cds_globals.\\gnd! "; *) gnd1;

    input (* integer supplySensitivity = "pwr1";
           integer groundSensitivity = "gnd1"; *) a;

    output (* integer supplySensitivity = "pwr1";
            integer groundSensitivity = "gnd1"; *) y;

    not il (.y(y), .a(a));

endmodule
```

Related Topics

- [supplySensitivity](#)
- [groundSensitivity](#)
- [Block-Based Discipline Resolution for Multiple Power Supply Design](#)
- [ie Statement in an amsd Block for Multiple Power Supply Design](#)
- [Multiple Power Supplies for Designs](#)
- [Interface Elements for Multiple Power Supply Designs](#)

Interface Elements for Multiple Power Supply Designs

AMS Designer Simulator allows selective insertion of individual interface elements (IEs) when one analog net is connecting to two or more digital ports that have different supply conditions.

You can instruct the Elaborator to insert separate interface elements (connect modules) using the `xrun/xmelab` command-line option, `-ams_check_msup`. You can split interface elements in any of the following design scenarios, where you have one net connecting to:

- Ports with different discrete domains in the `ie` card statements
- Ports with different supply-sensitive connect module supplies
- Child instances with different inherited connection as connect module supplies

Example 1

The following is an example of using the `-ams_check_msup` option for inserting separate interface elements for a design with different supply-sensitive connect module supplies, 3.3 and 1.2. The design is illustrated in the `test_res.vams` file. The tool inserts two interface elements with signal `testbench.net1` though it is using merge mode IE insertion.

```
xrun test_res.vams -amsconnrules CR_ss_full_fast -timescale 1ns/1ps -dres -input  
probe.tcl amscf.scs -ams_check_msup
```

```
//test_res.vams  
'include "disciplines.h"  
  
module mod3 (inpl, VDD, VSS);  
  
input VDD, VSS;
```

```
electrical VDD, VSS;
ground VSS;

input (* integer supplySensitivity="VDD";integer groundSensitivity="VSS"; *) inpl;
logic inpl;

endmodule

module mod4 (inpl, VDD, VSS);

input VDD, VSS;
electrical VDD, VSS;
ground VSS;

input (* integer supplySensitivity="VDD";integer groundSensitivity="VSS"; *) inpl;
logic inpl;

endmodule

module testbench();
    electrical gnd;
    ground gnd;
    wire net1;
    mod3 instance3 (net1, VDD, gnd);
    mod4 instance4 (net1, VDD2, gnd);
    vsource #(.wave({0, 0, 10u, 0, 20u, 2.2, 30u, 3.3, 70u, 3.3, 80u, 2.2, 90u, 0 })),
    .dc(3.3), .type("pwl")) V0 (net1, gnd);

    analog begin
        V(VDD) <+ 3.3;
        V(VDD2) <+ 1.2;
    end
endmodule

module cds_globals();
    electrical vdd , \vss! ;
endmodule

//amscf.scs
tran tran stop=100u
opt options save=all
```

Example 2

The following is an example of using the `-ams_check_msup` option for inserting separate interface elements for a design, with different disciplines, 3.3 and 1.2. The disciplines are specified in the `ie` card statements defined in the `amscf.scs` file.

The tool inserts two interface element with signal `testbench.net1` because the two connected ports have different supply value.

```
xrun clean test_res.vams -amsconnrules CR_full_fast -discipline logic amscf.scs
+dr_info -timescale 1ns/1ps -dres -ieinfo -clean -input probe.tcl -ams_check_msup

//test_res.vams
`include "disciplines.h"
module mod1 (inp1, VDD, VSS);

input VDD, VSS;
electrical VDD, VSS;
ground VSS;

input inp1;
...
endmodule

module mod3 (inp1, VDD, VSS);

input VDD, VSS;

electrical VDD, VSS;
ground VSS;

input inp1;

...
endmodule

module testbench();
    mod3 instance3 (net1, VDD, gnd);
    mod1 instance1 (net1, VDD, gnd);
endmodule

//amscf.scs
amsd {
    ie vsup=3.3  inst="testbench.instance1"
    ie vsup=1.2  inst="testbench.instance3"
}

simulator lang=spectre
tran tran stop=100u
opt options save=all
```

Related Topics

- [ie Statement in an amsd Block for Multiple Power Supply Design](#)
- [Block-Based Discipline Resolution for Multiple Power Supply Design](#)
- [Supply-Sensitive Modules for Multiple Power Supply Designs](#)

Mixed-Signal Design Elaboration

In the multi-step *xrun* invocation method, you must elaborate a design hierarchy before simulating the design. The `xrun -elaborate` command runs a language-independent elaborator, which does the following:

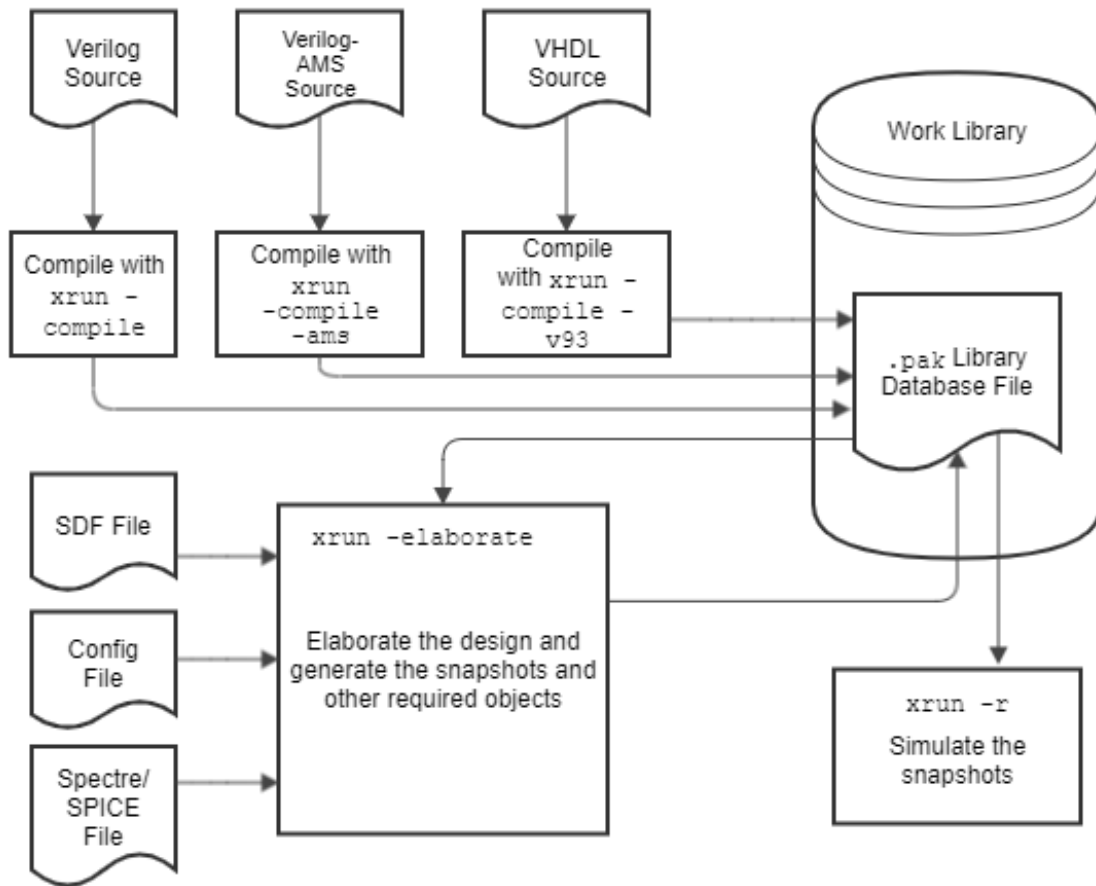
- Handles imported netlists such as Spectre and SPICE netlists
- Constructs a design hierarchy based on the instantiation and configuration information in the design
- Establishes signal connectivity
- Computes initial values for all objects in the design

The elaborator stores snapshots of your design hierarchy in the library database file along with other intermediate objects from compilation and elaboration. The simulator uses these snapshots during simulation. See also [Illustrating the xmelab Process](#).

The elaborator supports the occurrence-based binding feature of the Cadence [hierarchy editor](#).

Illustrating the xmelab Process

The following figure illustrates the elaboration (`xrun -elaborate`) process.



Related Topics

- [Using the Simulation Front End \(SFE\) Parser](#)
- [Binding during Elaboration](#)
- [Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)
- [Selecting a Delay Mode](#)
- [Setting Pulse Controls](#)

Access to Digital Simulation Objects

By default, the elaborator enables full access to digital simulation objects in the VHDL portion of a design. In addition, you always have full access to analog objects.

However, the elaborator marks all digital objects in a Verilog-AMS design as having no read or write access and disables access to connectivity (load and driver) information. Turning off these

three forms of access allows the elaborator to perform a set of optimizations that can dramatically improve the performance of the digital solver.

The only exceptions to this default mode are digital objects that are used as arguments to user-defined system tasks or functions. These objects are automatically given read, write, and connectivity access. By default, no access is given to objects that are used as arguments to built-in system tasks or functions. Using a construct that does not have a value (a module instance, for example) as an argument has no effect on access capabilities.

Generating a snapshot with limited visibility into simulation constructs and running the simulation in regression mode has significant performance advantages. However, turning off access to the HDL data structures imposes the following limitation: You cannot access simulation objects from a point outside the HDL code, through Tcl commands or through PLI.

You can turn on read, write, and connectivity access by using the following elaborator `xrun` command-line options:

- `-ACCEss`: Use this option to specify the access capability for all objects in the design.
- `-AFile`: Use this option to include an access file in which you specify the access capability for particular instances and portions of the design.

Related Topics

- [Access to Simulation Objects](#)
- [Using an Access File](#)
- [Guidelines for Access Control](#)

Binding During Mixed-Signal Design Elaboration

Binding is the process of selecting which design units are instantiated at each location in the design hierarchy. For mixed-signal designs, a design unit might be a module, [UDP](#), or analog (SPICE or Spectre) block. The elaborator binds each design unit that you instantiate in another, higher-level design block (such as a module) to a particular `Library.Cell:View`.

- The [default binding mechanism](#)
- The `xmelab -binding` option, which you can use to force the binding of a cell to a particular library and view
- The ``uselib` compiler directive, which lets you override the default binding mechanism and all command-line options

- The `xmelab -modelpath` option, which lets you specify SPICE or Spectre source files for the models to be used in a specified scope and in scopes below the specified scope

These binding mechanisms do not apply to the top-level modules that you specify on the command line.

For the AMS Designer simulator, the binding priorities are as follows (from highest to lowest):

1. The following subset of analog primitives from the Verilog-AMS LRM or any analog block (defined as a SPICE netlist or Verilog-A module) you specify using the `sourcefile` property in `amsd.scs`:
`resistor`
`capacitor`
`inductor`
`tline`
`vcvs`
`vccs`
2. 5x config rules
3. `-modelpath` setting

Related Topics

- [Elaboration Command-Line Options](#)
- [-modelpath](#)

Delay Modes in Mixed-Signal Designs

Delay modes let you alter the delay values specified in your models by using command-line options and compiler directives. You can ignore all delays specified in your model or replace all delays with a value of one simulation time unit. You can also replace delay values in selected portions of the model.

You can specify delay modes on a global basis or on a module basis. If you assign a specific delay mode to a module, all instances of that module simulate in that mode. The delay mode of each module is determined at elaboration time and cannot be altered dynamically.

Related Topics

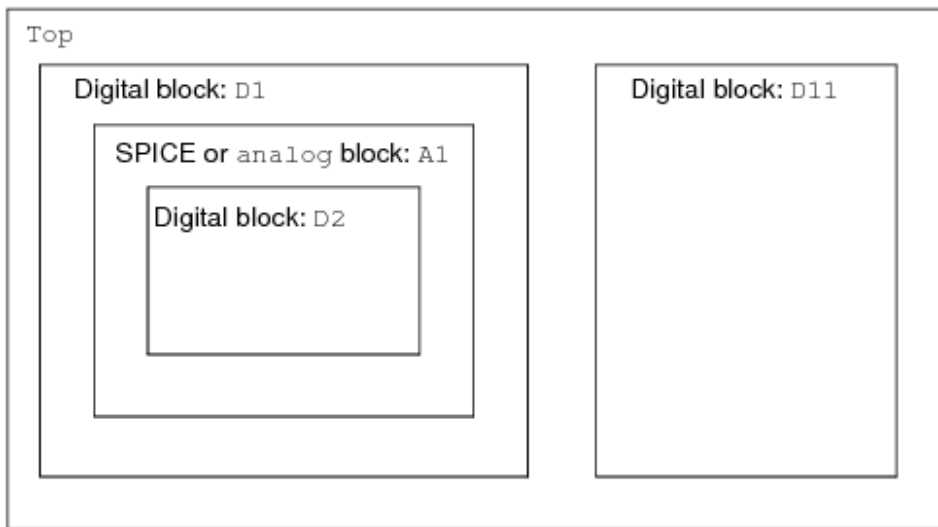
- [Delay Modes in Xcelium](#)
- [Setting a Delay Mode](#)

Guidance for Specifying Disciplines for Scopes

The several methods for specifying disciplines allow you to fine-tune the discipline resolution process. The following table lists how to apply these methods appropriately for different design configurations.

If the design...	Then...
Uses digital blocks with transistors at the leaf level	Turn off discipline resolution completely by specifying <code>-disres none</code> .
Has clearly defined boundaries between analog and digital	Turn off discipline resolution where it is not required by specifying <code>-setdiscipline "no_dr scope_discipline"</code> . Use <code>-disres default</code> or <code>-disres detailed</code> to specify the discipline resolution process to use elsewhere.
Does not have clearly defined boundaries between analog and digital	Run discipline resolution on the complete design and specify disciplines where required by specifying <code>-setdiscipline " scope_discipline "</code> . Use <code>-disres default</code> or <code>-disres detailed</code> to specify the discipline resolution process to use.

For example, you might have a design that contains a SPICE or Verilog-AMS block sandwiched between two digital blocks with known and clearly-defined boundaries between analog and digital.



Using ordinary discipline resolution without scopes, analog disciplines might propagate into domainless nets within the digital blocks. The following scoped options restrict such movement and allow the elaborator to insert connect modules at the well-defined boundaries.

If `top.D1.A1` is an analog behavioral block, such as Verilog-AMS, you might use the following options to specify a discipline of `electrical` for domainless nets in `A1` and to specify discrete disciplines for domainless nets in `D1` and `D2`.

```
-setdiscipline "no_dr inst-Top.D1- logic1"
-setdiscipline "no_dr inst-Top.D1.A1- electrical"
-setdiscipline "no_dr inst-Top.D1.A1.D2- logic2"
```

If `top.D1.A1` is a SPICE block introduced through an `amsd` block or a `modelpath` setting, there is no need to specify the discipline because the use of the `amsd` block or `modelpath` setting indicates that the SPICE or Verilog-A block is an `analog` block of the `electrical` discipline. In this case, you need to specify settings only for the digital blocks.

```
-setdiscipline "no_dr inst-Top.D1- logic1"
-setdiscipline "no_dr inst-Top.D1.A1.D2- logic2"
```

To turn off discipline resolution completely, you might use the `no_dr` value at the top scope of the design. You can achieve the same effect by using the `-disres none` option.

```
-setdiscipline "no_dr cell-Top- logic1"
-setdiscipline "no_dr inst-Top.D1- logic1"
-setdiscipline "no_dr inst-Top.D1.A1- electrical"
-setdiscipline "no_dr inst-Top.D1.A1.D2- logic2"
```

Turning off discipline resolution speeds up elaboration but leaves you with the responsibility of identifying the partitions that require connect module insertion.

Notes on Specifying Disciplines for Mixed Verilog-AMS/VHDL-AMS Designs

For the `INST` form,

```
-setdiscipline "INST-hierarchical_instance_name - discipline_name"
```

you can use `:` or `.` to delimit each unit in the `hierarchical_instance_name` such that the following two statements have the same effect:

```
-setdiscipline "inst-top.vh.ve.vh.ve- logic2"
```

```
-setdiscipline "inst-top:vh:ve:vh:ve- logic2"
```

When VHDL-AMS is the top-level instance, you can use `:` to represent the top-level entity (instead of its name). For example, if you had a Verilog-AMS instance, `ve_1`, in a VHDL-AMS top-level entity, `VHDL_TOP`, you specify the full path as `:ve_1` or `.ve_1`:

```
-setdiscipline "inst:-ve_1- logic2"
```

```
-setdiscipline "inst-.ve_1- logic2"
```

Scope matching rules depend on the language for each level of the hierarchy: For VHDL-AMS, scope matching rules are case-insensitive; for Verilog-AMS, scope matching rules are case-sensitive. In the following example, scope matching rules are case-insensitive because `vh` is a VHDL-AMS instance that contains Verilog-AMS instances `ve1` and `ve2`, so both of these paths are the same:

```
top.vh.VE1.vh2.VE2 // case-insensitive scope matching rules (vh is VHDL)
```

```
top.vh.ve1.vh2.ve2 // these two paths are the same
```

In the following example, scope matching rules are case-sensitive because the top-level unit is Verilog-AMS, so these two paths are not the same:

```
top.vh.VE1.vh2.VE2 // case-sensitive scope matching rules (Verilog-AMS top)
```

```
top.VH.ve1.vh2.ve2 // these two paths are not the same (vh is different from VH)
```

For the `CELL` form,

```
"CELL- lib_name. cell_name : view_name - discipline_name "
```

```
"CELL- lib_name. cell_name - discipline_name "
```

```
"CELL- cell_name - discipline_name "
```

You must use a dot character (`.`) between the library and cell names, and colon (`:`) to delimit the view name. (The colon and dot characters are only interchangeable for the `INST` form.) As with the `INST` form, scoping rules depend on the language of the parent design unit. For example, the

following two cell specifiers are the same because the parent design unit is VHDL-AMS, so case-insensitive scope matching rules apply:

```
WORKLIB.VHDL_ENTITY:ARCH // case-insensitive scope matching rules apply
```

```
worklib.vhdl_entity:arch // so these two cells are the same
```

The following two cell specifiers are not the same because the parent design unit is Verilog-AMS, so case-sensitive scope matching rules apply:

```
WORKLIB.VE_TOP:MODULE // case-sensitive scope matching rules apply
```

```
worklib.ve_top:module // so these two cells are not the same
```

Related Topic

- [Precedence of Discipline Specification Methods](#)

Precedence of Discipline Specification Methods

When disciplines are applied to nets in more than one way, the tool during elaboration determines the effective value by using the following rules of precedence.

Discipline Specification Method	Precedence
Explicitly defining the discipline. For example, <pre>module example2; electrical net; endmodule</pre>	Highest precedence
Overriding a discipline with out-of-module references (for domainless nets). For example, <pre>module example1; electrical example2.net; endmodule</pre>	
Defining disciplines with the <code>-setdiscipline</code> option (for domainless nets). For example, <pre>-setdiscipline "no_dr inst-top.d1- logic1"</pre>	

Obtaining disciplines through the discipline resolution process (for domainless nets).

Define discipline with `-setdiscipline` option together with `-ams_weak_setd` (for domainless nets).

Determining disciplines through application of default digital discipline specifications, including use of the ``default_discipline` compiler directive and the scope-based default digital discipline option (for nets in the discrete domain). For example,

```
`default_discipline logic;
```

used in a Verilog module and

```
xmelab -discipline logic
```

used on the command line, specify a global default logic discipline for domainless nets that you can override using methods shown higher in the chart.

The ``default_discipline` compiler directive is applied at analysis ( `xmvlog`) stage to the source on which it applies. During elaboration, since the source is already analyzed, there is no way to inherit the source-level settings from the instantiating scope. The same behavior exists for other compiler directives like ``default_nettype`. If you want to control the discipline resolution hierarchically, use the `-setdiscipline` option.

Related Topics

- [Guidance for Specifying Disciplines for Scopes](#)
- [Setting Block-Based Discipline Resolution Specifications in Search Libraries](#)

Setting Block-Based Discipline Resolution Specifications in Search Libraries

To apply block-based discipline resolution (BDR) to existing command-line-based search libraries use the `VLIB` and `YLIB` BDR directives.

The syntax for specifying the `VLIB` and `YLIB` BDR directives is:

```
xrun -v <library_file> and -y <library_directory>
```

They are used to define the BDR settings for components in these libraries. These extensions are supported in the `-setdiscipline` and `-scope_discipline` options in the following format:

Specification Type	Option Format
VLIB	<code>-setdiscipline "vlib-<library_file>- <discipline>"</code> <code>-scope_discipline "vlib-<library_file>- <discipline>"</code>
YLIB	<code>-setdiscipline "ylib-<library_directory>- <discipline>"</code> <code>-scope_discipline "ylib-<library_directory>- <discipline>"</code>

where `<library_file>` maps to the associated `-v <library_file>` and `<library_directory>` maps to the associated `-y <library_directory>`.

Wildcards are not supported for this feature.

Related Topics

- [Precedence of Discipline Specification Methods](#)
- [Guidance for Specifying Disciplines for Scopes](#)

Setting Pulse Controls for Mixed-Signal Designs

In the AMS Designer simulator and Xcelium simulator with the mixed-signal option, both module path delays and interconnect delays are simulated as transport delays by default. There is no command-line option to enable the transport delay algorithm. You must, however, set pulse control limits to see transport delay behavior. If you do not set pulse control limits, the software sets them equal to the delay by default; no pulse having a shorter duration than the delay passes through.

To set global pulse limits for both module path delays and interconnect delays, use the `-pulse_r` and the `-pulse_e` options when you invoke the elaborator.

Full pulse control is available for both types of delays. You can:

- Set global pulse control for both module path delays and interconnect delays
- Set global pulse limits for module path delays and interconnect delays separately in the same simulation
- Narrow the scope of module path pulse control to a specific module or to particular paths within modules using the `PATHPULSE$` specparam
- Specify whether you want to use On-Event or On-Detect pulse filtering

Related Topics

- [Setting Pulse Controls](#)
- [Setting Global Pulse Control](#)

Simulation Front End (SFE) Parser

The AMS Designer simulator uses the simulation front end (SFE) analog parser. With the SFE parser, the Spectre solver of the AMS Designer simulator use the same analog parser as the Spectre circuit simulator. The SFE parser provides enhanced performance and additional features, including the option to run in 64-bit mode.

If you currently use the AMS Designer simulator with the old Spectre parser and are upgrading from a version prior to IUS 6.11, you need to review important information about [Migrating from the Old Spectre Parser](#) so that you can evaluate the differences between the old Spectre parser and the SFE parser carefully. If you have any questions, contact Cadence.

The SFE parser provides:

- Native support for Spectre, SPICE, and HSPICE input files, including Spectre netlist compiled functions (NCFs)

See "Netlist Compiled Functions (NCFs)" in the *Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide* for more information.

- Simplified input commands with no need to run the Spectre preprocessor (see [Using Simplified Input Commands with the Simulation Front End Parser](#) for an example)
- Support for elaborating and simulating top-level instances and analyses in model files and analog control files
- Support for instances of structural Verilog-A included using an `ahdl_include` statement in SPICE and Spectre blocks (see also [Including Structural Verilog-A in a Spectre Netlist](#))
- Support for hierarchical identifiers that allow you to connect to an internal node of a subcircuit
- Support for SPICE-syntax identifiers so that instance, node, and parameter names can contain characters such as `#`, `@`, and `|`
- Support for several global node statements in a design
- Support for mixed Spectre and SPICE syntax You can include both the Spectre and SPICE languages in a design, as long as you insert the appropriate simulator language switch (`lang`). The parser checks SPICE language syntax to verify compliance with language requirements.
- Support for SPICE-syntax model binning so that you can bin models according to geometry and size
- Support for compiled C flow to boost performance, particularly when you use Verilog-A to model CMOS devices such as MOSFETs, resistors, and capacitors. The software compiles Verilog-A modules and `bsource` devices that you include in a Spectre or SPICE file using an `ahdl_include` statement during simulation.
- Support for the AMS Designer simulator in your verification flow
- Support for SPICE and Spectre user-defined functions only on analog instances inside Verilog-AMS modules (see [Using SPICE and Spectre User-Defined Functions](#))
- Support for paramsets (for Verilog-A only) Paramsets provide a convenient way to collect parameter values so that a particular instance need only specify overrides specifically

required for that instance.

- Consistent waveform formatsThe SFE parser uses the [SST2](#) format for waveforms you request using both Tcl probes and analog control file statements. You can use the `rawfmt` option to output data in other formats such as `wdf` or `fsdb`.

While the software changes the names of Verilog-AMS modules, nets, and variables belonging to analog behavioral blocks, you can continue to reference these objects using their original names in Tcl `probe` and `value` commands.

Including Structural Verilog-A in a Spectre Netlist

Using the AMS Designer simulator with the simulation front end (SFE) parser, you can include instances of structural Verilog-A in SPICE and Spectre blocks using an `ahdl_include` statement. A structural module is one that instantiates another module.

The following file (`res.va`) contains module `res` and module `vastruct`. Because module `vastruct` instantiates module `res`, it is a structural module.

```
// res.va -- Verilog-A file containing a structural module, vastruct

`include "discipline.vams"
`include "constants.vams"

module res(vp, vn);
  inout vp, vn;
  electrical vp, vn;
  parameter real r = 0;
  analog begin
    V(vp, vn) <+ r*I(vp, vn);
    $display("Verilog-A resistor:\n");
    $display("Voltage=%f, Current=%f\n", V(vp, vn), I(vp, vn));
  end
endmodule

module vastruct(p, n);          // structural Verilog-A module
  inout p; electrical p;
  inout n; electrical n;
  parameter real r=0;
  res #(.r(r)) Rva (p, n); // instantiates module res
endmodule
```

You can include the file containing the structural module in a Spectre netlist file using the `ahdl_include` statement as follows:

```
global gnd
simulator lang=spectre
ahdl_include "res.va" // Includes a structural Verilog-A module

subckt sub (p n)
parameters r=500
    Rsp p int1 resistor r = r
    Rva int1 n vastruct r=r // Instantiates the structural Verilog-A module
ends
```

You can elaborate this design structure using the SFE parser.

Using SPICE and Spectre User-Defined Functions

Using the AMS Designer simulator with the simulation front end (SFE) parser, you can have SPICE and Spectre user-defined functions on analog instances inside Verilog-AMS modules. For example:

```
module foo
...
    vsource #(.dc(spiceUDF(5.0)) v1(in, out);
        // spiceUDF is a user-defined function inside SPICE
...
endmodule
```

You might use such a function in the calculation of instance parameter values such as `ps`, `pd`, `ad`, or `as`. For example, you might define a function `f_mod` as

```
real f_mod(real a, real b) {
    return (a-b*int((a+0.5)/b)) ;
}
```

and use this function in the calculation of instance parameter `ad` like this:

```
ad=f_mod*iPar("l")
```

You cannot mix a user-defined function with a digital function in an expression.

Using Simplified Input Commands with the Simulation Front End Parser

Using the AMS Designer simulator with the simulation front end (SFE) parser, you do not need to run the Spectre preprocessor and the input commands are simpler.

An example of a simplified elaboration command is:

```
xmelab -MODELPATH subckts.m
```

Migrating from the Old Spectre Parser

If you were using the old Spectre parser in a previous release, you should be aware of the following differences when using the simulation front end (SFE) parser, which is turned on by default:

- The SFE parser creates instances of `_cds_internal_modules_` for analog blocks.
- The SFE parser publishes global signals from `cds_globals` to the top scope using names that are different from those that the old parser used.
- Supply sensitivity values do not propagate to connect modules when you use the SFE parser.
- The following VHDL-AMS features are not available when you run the AMS Designer simulator with the SFE parser:
 - The `-delta` option to the Tcl `run` command
 - Break statements using break lists
 - Configuration declarations
 - The `===` operator

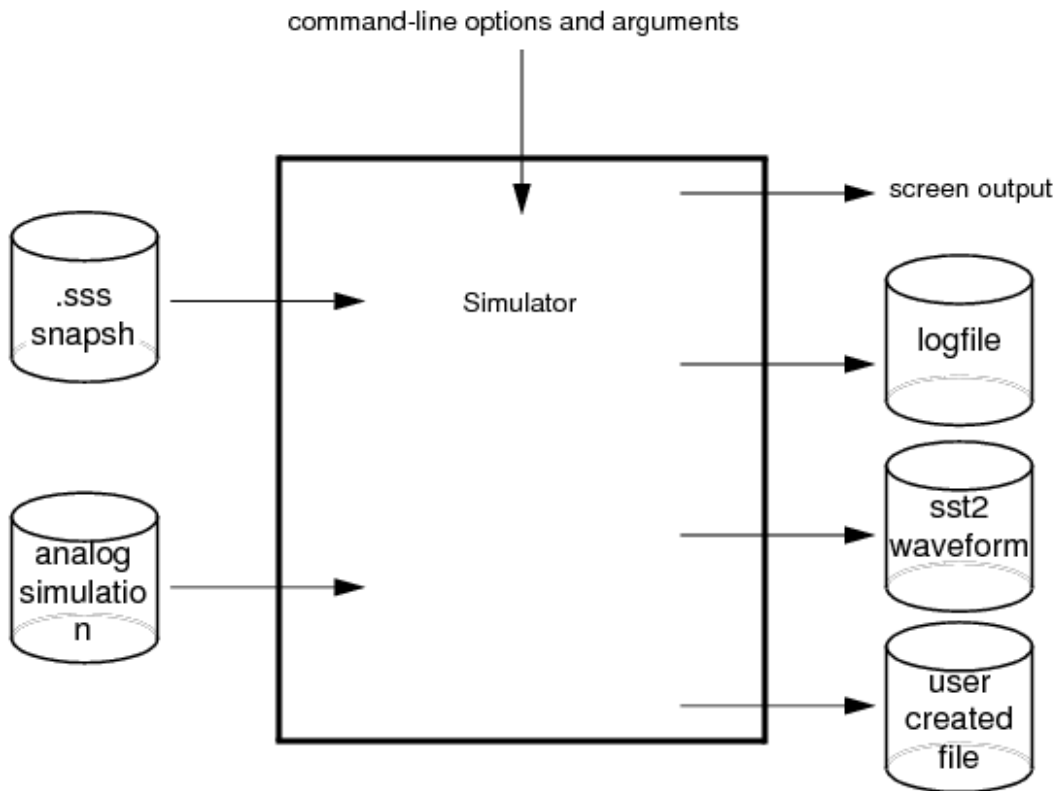
Mixed-Signal Design Simulation

After you have compiled and elaborated your design, you can simulate the design using `xrun -R`. This program uses the compiled-code streams to simulate the dynamic behavior of the design.

The `xrun` utility loads the snapshot as its primary input. It then loads other intermediate objects referenced by that input. In the case of interactive debugging, HDL source files and script files also might be loaded. Other data files might be loaded (via `$read*` tasks or `textio`) if the model being simulated requires them.

You control the outputs of simulation using the model, the analog simulation control file, or the debugger. Result files can come from the model, Simulation History Manager ([SHM](#)) databases, or Value Change Dump ([VCD](#)) files.

The following figure illustrates some of the inputs and outputs that the simulator can use and generate:



Related Topics

- [XRUN Invocation for Mixed-Signal Designs](#)
- [Running the Simulator](#)

Running the Simulator

You can run the Xcelium simulator (`xrun -R`) in:

- **Noninteractive mode:** Automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input.
- **Interactive mode:** Stops the simulation at time 0 and returns the `xcelium>` prompt.

In either mode, you can run the simulator with or without the SimVision environment.

Starting or Resuming a Simulation

To start or resume a simulation,

- If you are using the Tcl command-line interface, use the `run` command with the appropriate option to control when you want the simulation to stop.
- If you are using the SimVision environment, use the commands on the Simulation menu or the buttons on the simulation control toolbar.

Related Topics

- [Controlling the Simulation](#)
- [run](#)
- [Mixed-Signal Design Simulation](#)
- [Exiting the Simulation](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)
- [Process-Based Save-and-Restart](#)

Restarting the Simulator from a Previously-Saved Snapshot

Using the multi-step `xrun` invocation method, you compile your modules using `xrun -compile`, then you elaborate using `xrun -elaborate`, which elaborates only newly-compiled modules: If you have not recompiled a module, the elaborator does not re-elaborate it. The third step is to simulate using `xrun -R`.

When you simulate using a previously-saved snapshot, the simulator (`xrun -r`) verifies that the snapshot depends on modules that have a time stamp that is earlier than the snapshot itself. You can run the simulator from a previously-saved snapshot as indicated by the following use models:

Use Model 1 (Warm Restart)

1. Compile and elaborate to generate snapshot `A`.
2. In SimVision, simulate using snapshot `A` for some time (such as 1000 ns).
3. Save snapshot `B` (`save :B`).
4. Exit simulation.
5. Re-elaborate the design to generate snapshot `A`.

You can run `xrun -R` using snapshot `B`. If you run `xrun -R` using snapshot `A` at this point, you cannot use `Tcl restart` | [restart](#) to simulate using snapshot `B` because snapshot `A` now has a newer timestamp than snapshot `B`. You can simulate snapshot `B` only from the `xrun -R` command line.

Use Model 2 (Cold Restart)

1. Compile and elaborate to generate snapshot `A`.
2. In SimVision, simulate using snapshot `A` for some time (such as 1000 ns).
3. Save snapshot `B` (`save :B`).
4. Exit simulation.
5. Re-elaborate the design to generate snapshot `C`.

You can run `xrun -R` using snapshot `B`. You can run `xrun -R` using snapshot `A`. You can also use `Tcl restart` to simulate using snapshot `B`. However, if you run `xrun -R` using snapshot `C` at this point, you cannot use `Tcl restart` to simulate using snapshot `B` because snapshot `C` now has a newer time stamp than snapshot `B`. You can simulate snapshot `B` only from the `xrun -R` command line.

Related Topics

- [Using the Save-and-Restart Feature](#)
- [Mixed-Signal Activity Statistics](#)
- [Running the Simulator](#)
- [Exiting the Simulation](#)

Using the Save-and-Restart Feature

In mixed-signal designs, you can save a snapshot of a simulation and restart after making changes to simulation parameters and models. When you save a simulation with process-based save/restart, you create a snapshot of the entire simulation process state (memory, file pointers, and so on), including all integrated C/C++ code, third-party applications, system function \$save in SV/Verilog/VHDL module, external codes such as System C in the design, all allocated stacks, heaps to disk.

After you save the simulation, you can either do a cold restart, or a warm restart.

A cold restart refers to a situation in which the simulator is (re-)invoked with the saved snapshot. This approach provides an opportunity to change certain simulator invocation option settings.

A warm restart refers to a situation where the simulation runs in the same directory as the saved simulation. Essentially, the restarted simulation is presented as a continuation of the existing session. Since the simulator is not explicitly re-invoked, there is no opportunity to change any simulator invocation option settings. They remain the same as they were at the time of the save operation. Warm restart is most commonly used for debugging purpose.

For AMS designs using the analog solvers such as Spectre solver, APS solver, or XPS-MS solver, you can use save-and-restart either in non-interactive mode (command-line) or in Tcl mode.

You can also use the save-and-restart feature in mixed-signal designs containing SystemC models. The use model is the same in terms of the usage of the Tcl commands save and restart. However, in the Tcl restart flow for SystemC, one difference from the traditional save-and-restart feature is that the analog netlist/model files are re-parsed and the analog circuit is re-elaborated so that you can update these files prior to specifying the restart Tcl command, and expect these changes to take effect in the restart simulation.

The save-and-restart feature for mixed-signal designs with SystemC does not work on SLES11.

Process-Based Save-and-Restart for Mixed-Signal Designs

You can use the process-based save-and-restart approach for AMS and DMS simulations. The approach is the same in terms of the usage of the Tcl commands `save` and `restart`. However, the Tcl restart flow for process-based save-and-restart differs from the traditional mixed-signal save-and-restart feature. In the process-based save-and-restart approach, during simulation restart, a new `xmsim` process is invoked, and the simulation state is restored to exactly the state when the save occurred.

You can enable process-based save-and-restart in the checkpoint enable mode and process save mode.

In the checkpoint enable mode, during the restart process, the original analog solver thread is terminated. The tool then relaunches a new analog solver, and the following processing are performed:

- The analog solver is re-initialized
- The sync between digital and analog solvers is re-established
- The analog circuits are re-elaborated
- The saved analog data is reloaded from the save directory
- The probes on analog signals are re-installed.

To enable checkpoint enable mode, you must use the the `-checkpoint_enable` command-line option with `xmsim/xrun`.

For AMS design simulations, it is recommended that you use the process save mode.

In the process save mode, the threads are handled by checkpoint package and multiple threads are supported. And, the analog engine running in child threads can be recovered. However, in the checkpoint enable mode, the analog engine needs to be restarted and re-initialized.

To enable the process save mode, you must use the `-process_save` command-line option with `xmsim/xrun`.

By default, it is recommended to run the simulations in the 64-bit mode.

Examples

The following examples show how you can enable process save mode save-and-restart flow in both a warm restart and cold restart:

Warm restart:

```
xrun top.vams receiver.vams amscf.scs -input probe.tcl -amsconnrule CR_full -clean -
```

process_save

The following example details the `probe.tcl` file used:

```
-----probe.tcl-----
database -open waves -into waves.shm -default
probe -create -all -depth all -waveform -database waves run 2ns
scope -describe top
scope -describe top.RE1
force top.RE1.scalar_out = 1'b0
value top.scalar_out top.RE1.scalar_out
save -overwrite save1
run 2ns
release top.RE1.scalar_out
value top.scalar_out top.RE1.scalar_out
run 2ns
restart save1
value top.scalar_out top.RE1.scalar_out
run 2ns
exit
```

Cold restart:

```
xrun top.vams receiver.vams amscf.scs -input save.tcl -amsconnrule CR_full -clean -
process_save
```

The following example details the `save.tcl` file used:

```
-----save.tcl-----
database -open waves -into waves.shm -default probe -create -all -depth all -waveform -
database waves run 2ns
scope -describe top
scope -describe top.RE1
force top.RE1.scalar_out = 1'b0
value top.scalar_out top.RE1.scalar_out
save -overwrite save1
run 2ns
release top.RE1.scalar_out
value top.scalar_out top.RE1.scalar_out
run 2ns
exit
```

The following is an example of the `.vams` and `amsd` control files that are:

```
-----top.vams-----
`timescale 1ns/1ps
```

```
`include "disciplines.vams"
module top();
  electrical scalar_in;
  electrical scalar_out;
  electrical [1:0] array_in;
  electrical [1:0] array_out;
  ///instantiation/////
  receiver RE1 (scalar_in,scalar_out,array_in,array_out);
  analog begin
    V(scalar_in) <+ 1.4;
    V(array_in[1]) <+ 1.8;
    V(array_in[0]) <+ 0.2;
  end
endmodule

-----receiver.vams-----
module receiver (scalar_in,scalar_out, array_in,array_out);
  input scalar_in;
  output scalar_out;
  input [1:0] array_in;
  output [1:0] array_out;
  logic scalar_in,scalar_out;
  logic [1:0] array_in,array_out;
  assign scalar_out = 1'b1;
  assign array_out= 2'b10;
endmodule

-----amscf.scs-----
Simulator lang=spectre
Tran tran stop=20ns
```

And, the following command invokes the restart command for a cold restart:

```
xrun -r save1 -input restart.tcl -process_save
```

Where, the restart.tcl file contains:

```
-----restart.tcl-----
value top.scalar_out top.RE1.scalar_out
run 2ns
exit
```

Related Topics

- [get_analog_param](#)
- [set_analog_param](#)
- [Restarting the Simulator from a Previously-Saved Snapshot](#)

Mixed-Signal Activity Statistics

After exiting a simulation, AMS Designer displays the mixed-signal activity statistics along with the additional activity statistics of the Interface Elements (IEs) for the simulation run.

Example:

```
**** AMSD: Mixed-Signal Activity Statistics ****

Number of A-to-D events:                857

Number of A-to-D events in IEs:         857

Number of D-to-A events:                138

Number of D-to-A events in IEs:         138

Number of VHDL-AMS Breaks:              0

Number of direct digital accesses:       56

**** AMSD: IE Additional Activity Statistics ****

Number of IEs with D-to-A X event:       3
Number of IEs with D-to-A Z event:       0
Number of IEs with A-to-D X event:       0
Number of IEs with A-to-D Z event:       0
Number of IEs with Vin exceeded limit:   4
Number of IEs with Vdd exceeded limit:   0
IE with max number of D-to-A conversions: 120
PLL_160MHZ_sim.CLK_160MHZ__L2E_2__logic
IE with max number of A-to-D conversions: 354
PLL_160MHZ_sim.I3.VCO_CLK__E2L_2__logic
Number of reporting IEs / total IEs:     8/8
Number of IEs with Vin/Vdd limit checked: 8
```

The following table describes the statistics displayed in the example given above.

Activity Statistic	Description
Activity Statistic	Description
Number of A-to-D events	The number of analog-to-digital events.
Number of A-to-D events in IEs	The number of analog-to-digital events in the IEs.
Number of D-to-A events	The number of digital-to-analog events.
Number of D-to-A events in IEs	The number of digital-to-analog events in the IEs.
Number of VHDL-AMS Breaks	The number of VHDL-AMS breaks.
Number of direct digital accesses	The number of direct digital accesses.
Number of IEs with D-to-A X event	The number of IEs that detected digital-to-analog events with x states.
Number of IEs with D-to-A Z event	The number of IEs that detected digital-to-analog events with z states.

Number of IEs with A-to-D X event	The number of IEs that detected analog-to-digital events with x states.
Number of IEs with A-to-D Z event	The number of IEs that detected analog-to-digital events with z states.
Number of IEs with Vin exceeded limit	The number of IEs for which the value of V_{in} exceeded the specified limit.
Number of IEs with Vdd exceeded limit	The number of IEs for which the value of V_{DD} exceeded the specified limit.
IE with max number of D-to-A conversions	The maximum number of digital-to-analog conversions for the IE mentioned below this statistic.
<i>PLL_160MHZ_sim.CLK_160MHZ__L2E_2__logic</i>	The name of the IE that has the maximum number of digital-to-analog conversions.
IE with max number of A-to-D conversions	The maximum number of analog-to-digital conversions for the IE mentioned below this statistic.

<code>PLL_160MHZ_sim.I3.VCO_CLK__E2L_2__logic</code>	The name of the IE that has the maximum number of analog-to-digital conversions.
Number of reporting IEs / total IEs	The number of reported IEs along with the total number of IEs in the design.
Number of IEs with Vin/Vdd limit checked	The number of IEs for which the <code>Vin</code> or <code>VDD</code> limit is checked.

Related Topics

- [Restarting the Simulator from a Previously-Saved Snapshot](#)
- [Using the Save-and-Restart Feature](#)

Other Tasks During Simulation

Some of the other tasks you can perform when you invoke the simulator are:

- [Updating Design Changes When You Run the Simulator](#): After editing a design unit, you can use `xrun -r -update` to recompile and re-elaborate all out-of-date design units before re-simulating.
- [Providing Interactive Commands from a File](#): If you want to load a file containing simulator commands (perhaps Tcl commands or aliases, or a key file containing commands saved from a previous simulation run), you can specify the input file with the `-input` option when you invoke the simulator. For example, `xrun -r -input <filename>`.

Related Topics

- [Invoking the Simulator](#)
- [Running the Simulator](#)

Exiting the Simulation

To exit the simulator at any time, perform one of the following:

- If you are using the Tcl command-line interface, type `exit` or `finish`. The exit command is a built-in Tcl command. It halts the execution and returns the control to the operating system.

The `finish` command also halts execution and returns control to the operating system. This command takes an optional argument that determines what type of information is displayed after exiting.

- 0 Prints nothing (same as executing `finish` without an argument).
 - 1 Prints the simulation time.
 - 2 Prints simulation time and statistics on memory and CPU usage.
- If you are using the SimVision environment, choose *File - Exit* or type `finish`.

If you type `finish`, the window disappears before you can read the information. However, the information also appears in the log file.

When the simulation encounters an `exit` command during transient analysis, the other analyses that are specified after the transient analysis (through analog control file) continues to run without any problem.

Related Topics

- [exit](#)
- [finish](#)
- [Running the Simulator](#)
- [Other Tasks During Simulation](#)

Specifying a Host for Spectre

To run Spectre on a local host:

- Use the `xrun` command with the `-analogsolver` command-line option.

Example:

```
xrun -analogsolver "spectre"
```

AMS Designer runs the `spectre` executable on the local host. This is the default behavior.

Alternatively, you can use a script or a resource scheduler, provided it boots Spectre.

To run Spectre on a remote host:

- Use the `bsub` command.

```
xrun -analogsolver "bsub -R \"OSNAME==Linux &7 OSREL==EE70 && SFIPLATFORM==c0700\"  
-n 1 -q lnx64 -J spectre"
```

Related Topic

[Managing Analog Resource Usage During Interactive Simulations](#)

Managing Analog Resource Usage During Interactive Simulations

When running interactive simulations with Spectre AMS Designer and reserving CPU resources with distributed resource packages like LSF, SunGrid or other resource management tools, though Spectre may be inactive when the simulation is stopped, the managed resources are still allocated. The only way to keep the interactive AMS simulation running while freeing the associated resources is to exit Spectre.

To do this:

1. Invoke Spectre on the `xrun` command line using the `-analogsolver` option.
This starts the interactive simulation.
2. Exit Spectre using the `exit -analogsolver Tcl` command. This option lets you exit Spectre and triggers a Process-Based Save and Restart (PBSR) Spectre snapshot so that you can recall the analog solver if you want to simulate again.
The resource scheduler releases the resources reserved for Spectre and they become available to other users.

To restart the simulation, use the `run -analogsolver` TCL command. Spectre is booted, the state is restored and the simulation may continue.

You can also run Spectre on a different CPU core and control where it is booted.

Examples:

Runs Spectre on the current CPU core.

```
xrun -analogsolver spectre
```

Runs Spectre on the specified CPU core.

```
xrun -analogsolver "[apsub --jobclass xcelium --slots 8 -x 10d --cmd] spectre +mt=8  
+preset=lx"
```

Related Topics

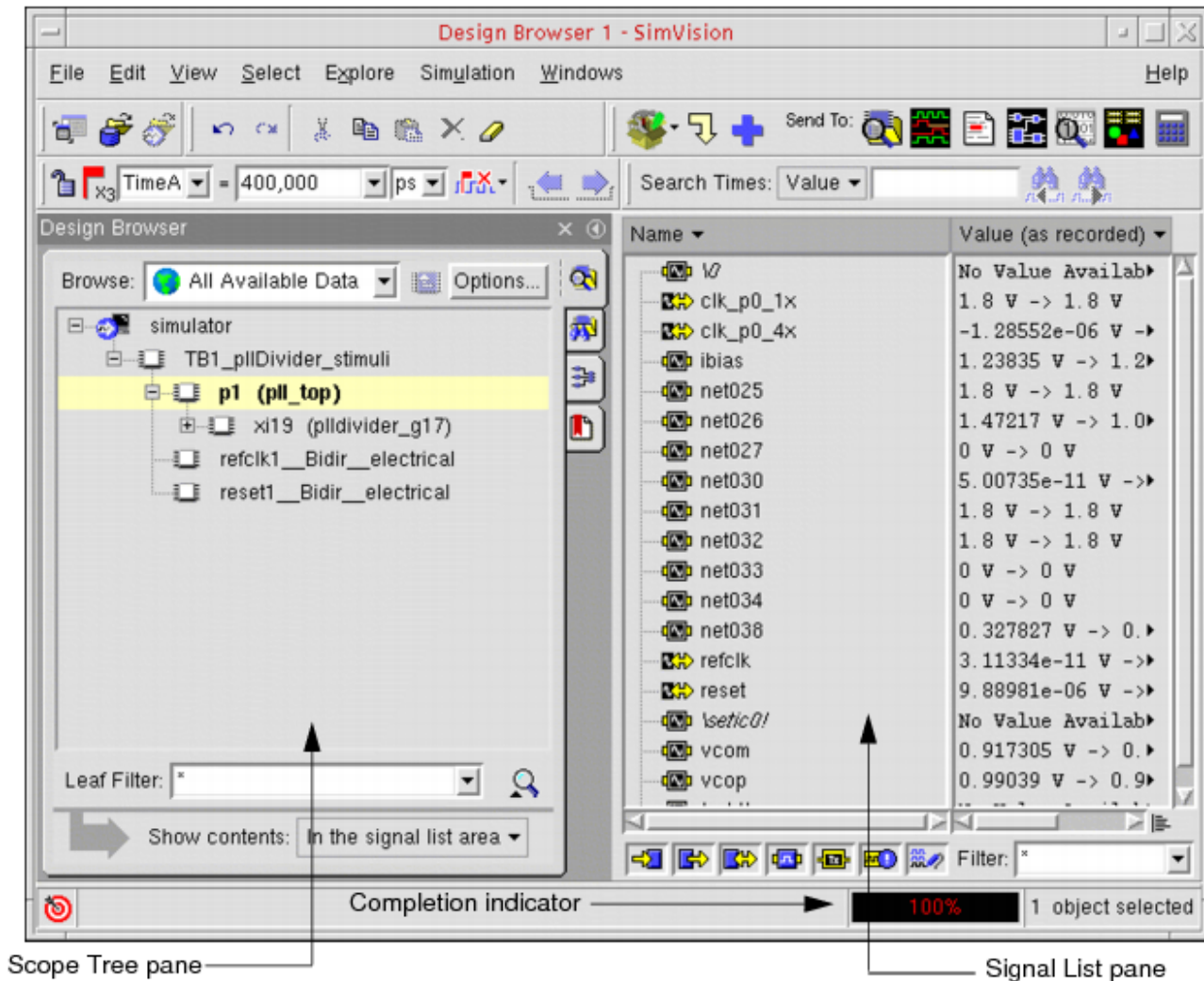
- [exit](#)
- [run](#)
- [analogsolver](#)
- [Specifying a Host for Spectre](#)

SimVision with AMS Simulator

When you use the SimVision environment with the Spectre®AMS Designer simulator, you have access to additional controls to support Verilog®-AMS and other AMS-related design units. The SimVision environment appears when you use the `-gui` option with the `xrun` or `xmsim` command or when you run the `simvision` command (either from the command line or from a Tcl file that you specify using the `-input` option to the `xrun` or `xmsim`).

The Design Browser Window for AMS Designs

The Design Browser presents a graphical display of your design and provides access to the other SimVision windows. The Design Browser for the AMS Designer simulator looks something like the following illustration:



The Design Browser contains two primary panes. On the left is the Scope Tree pane, which displays your current design hierarchy in a graphical tree representation. On the right is the Signal List pane, which displays a list of signals with their current simulation values. The completion indicator appears at the bottom of the window.

The SimVision environment can also read PSFXL analog simulation data. It supports a unified PSFXL/SST2 database that contains both SST2 digital data and PSFXL analog data, so that you do not need to manually segregate analog probes in a separate database when probing or opening databases to view the results.

Starting with the INCISIV 14.1 release, the PSFXL/SST waveform data can also be read using the Virtuoso Visualization and Analysis (ViVa) tool in standalone mode.

The unified PSFXL/SST2 database stores the results in a sub directory, with two containers. The digital signal results are stored in a SST2 container (*.trn) and the analog signal results are stored

in an analog container (*.tran). The `logFile`, created during the simulation, links these two containers in a unified database.

The Design Browser lets you:

- Select branches in addition to the objects available for a Verilog module by choosing *Select - Branches*

In the Source Browser, the left parenthesis enclosing each branch appears highlighted, as illustrated here:

```
// A digital event to which analog is made sensitive to
@(posedge(trigger))
    value = V(inSig);

V(holdSig) <+ transition( value, 1n, 2.5n);
```

- Navigate through the edges by using the *Previous Edge* and *Next Edge* buttons. These apply to only digital signals



- Open the Source Browser by clicking the *Source Browser* icon.



The text view of the components in your design appears in the Source Browser window.

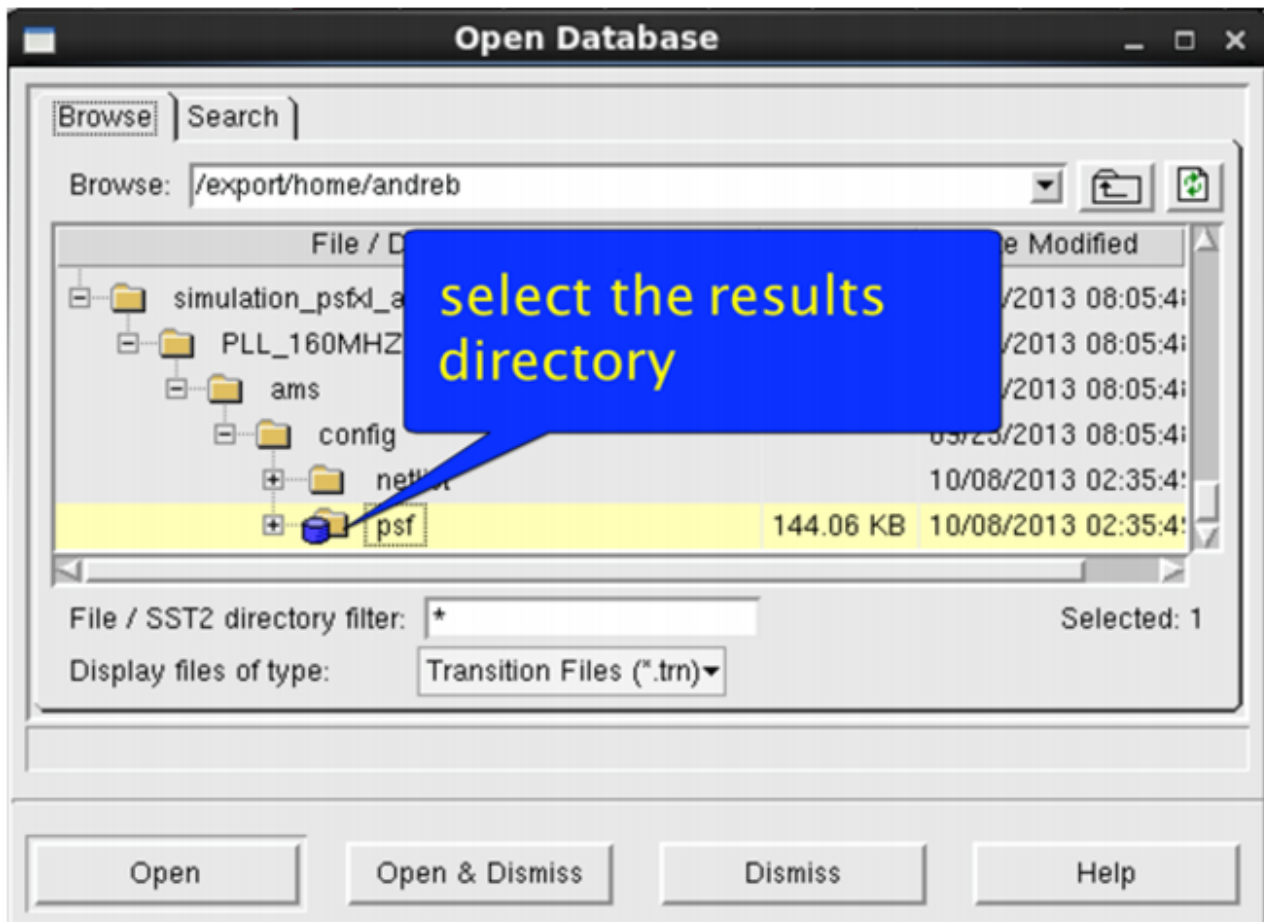
Related Topics

- [Invoking SimVision](#)
- [SimVision Window Menu Bar](#)
- [Using the Source Browser](#)
- [Viewing Analog Data in the Waveform Viewer](#)

Invoking SimVision

It is recommended that you invoke SimVision from a directory other than the unified PSFXL/SST2 database result directory so that you can directly open the unified PSFXL/SST2 database with both the containers. This can be done as follows:

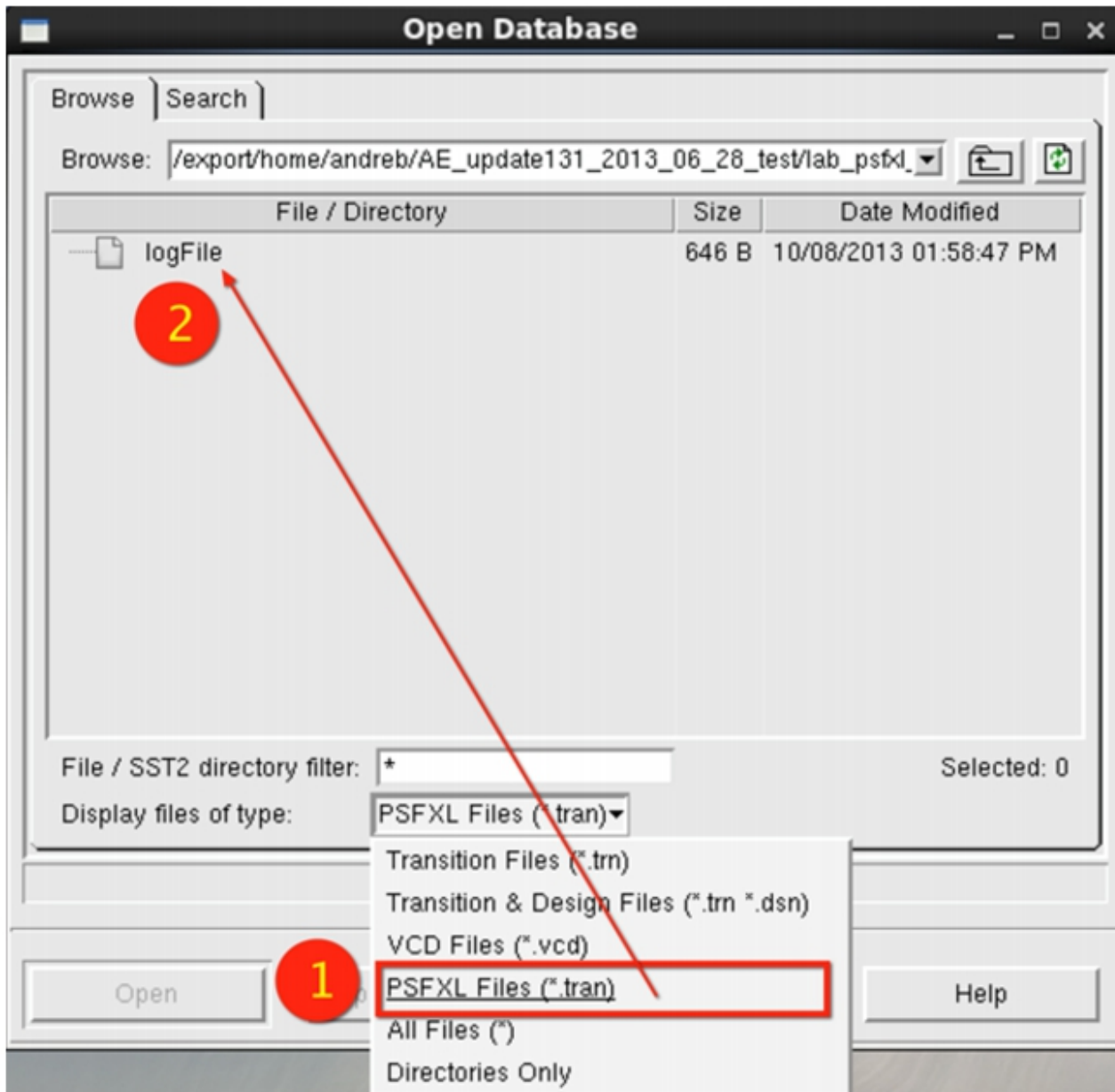
1. Select the *File - Open Database* menu option, or click the *Open an SST2 database* icon. The Open Database window is displayed, as shown below.



2. Select the directory containing the results.
3. Click *Open & Dismiss*.

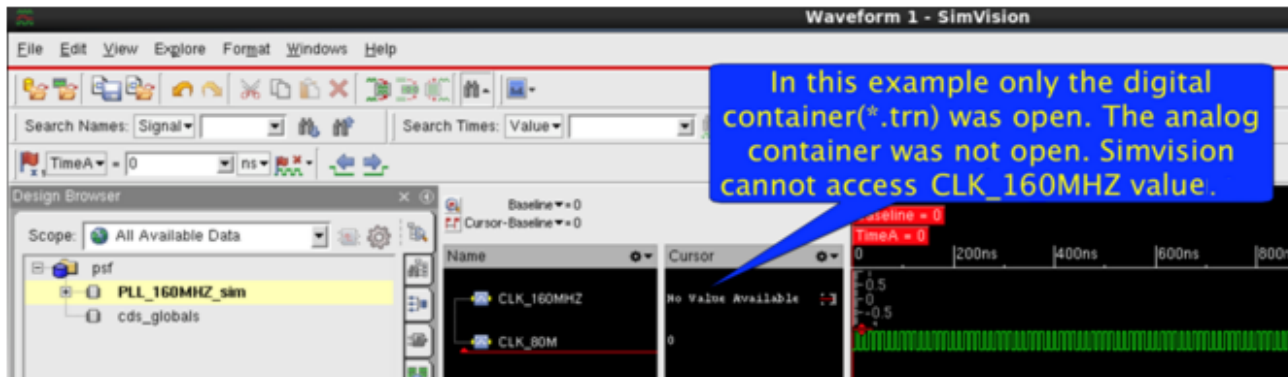
The unified database containing both the containers is displayed in SimVision.

To display the unified database containing only the analog container in SimVision, expand the results directory, choose *PSFXL Files (*.tran)* from the *Display files of type* drop-down list, select *logFile* (created during simulation), and click *Open & Dismiss*.



The unified database containing only the analog container is displayed in SimVision.

Though it is possible to open only the simulation digital (*.trn) container results during post-processing by selecting *Transition Files (*.trn)* from the *Display files of type* drop-down list, it is not recommended, because in this case, SimVision is not able to plot the analog vector values that are stored in the analog container. For such cases, SimVision displays *No Value Available*, as shown below.



Related Topics

- [SimVision Window Menu Bar](#)
- [Plotting Signals in the Waveform Window](#)

SimVision Window Menu Bar

To support Verilog-AMS, the menu choices in the Design Browser (and in the other SimVision windows) for the AMS Designer simulator differ from the choices available for purely digital simulators. See the comments and cross-references in the following table.

Menu	Command	Description
<i>Edit</i>		Lets you specify the options in the Preferences form.
	<i>Preferences</i>	Opens the Preferences form and lets you set the display and formatting preferences for Verilog-AMS objects. For more information, see Setting Display and Formatting Preferences for Verilog-AMS Objects .
<i>Select</i>		Lets you select the objects and branches for a Verilog module.

	<i>Branches</i>	Lets you select branches in addition to the objects available for a Verilog module. For more information, see Selecting Objects. To probe the values of currents and Spectre primitives, you must use Tcl probe commands.
<i>Simulation</i>		Lets you specify the simulation options for SimVision.
	<i>Advance - To Synchronization Point</i>	Runs the simulator until the digital solver gains control from the analog solver. You can use this command to leave the analog solver and return to the digital solver. (Some commands, such as <i>Create Force</i>, are available only when the digital solver is active.) The simulator stops at any breakpoint it encounters before it reaches the synchronization point.
	<i>Advance - Timepoint</i>	Simulates up to the beginning of the next analog timepoint or to the next time at which a digital event is scheduled. The simulator stops at any breakpoint it encounters before it reaches the specified timepoint.
	<i>Reset to Start</i>	Supported for RNM designs. For more details of that feature, please refer to <i>SimVision User Guide</i> .
	<i>Set Breakpoint - Time</i>	For the AMS Designer simulator, you can specify a time breakpoint at any time: You do not have to use an integer to specify the number of time units.

	<i>Deposit Value</i>	You cannot deposit values to analog quantities, including analog nets, ports, variables, or branches. Also, you cannot deposit values to digital quantities while the analog solver is active.
	<i>Create Probe</i>	Probe values of digital and most analog objects to a database. To probe the values of currents and Spectre primitives, you must use Tcl probe commands.
	<i>Create Force</i>	Only available when the digital solver is active. You cannot force a value for an analog quantity, including analog nets, branches, and analog variables. Also, you cannot force values for digital variables and signals while the analog solver is active.

Related Topics

- [The Design Browser Window for AMS Designs](#)
- [Using the Menus and Forms for AMS Designs](#)
- [Plotting Signals in the Waveform Window](#)

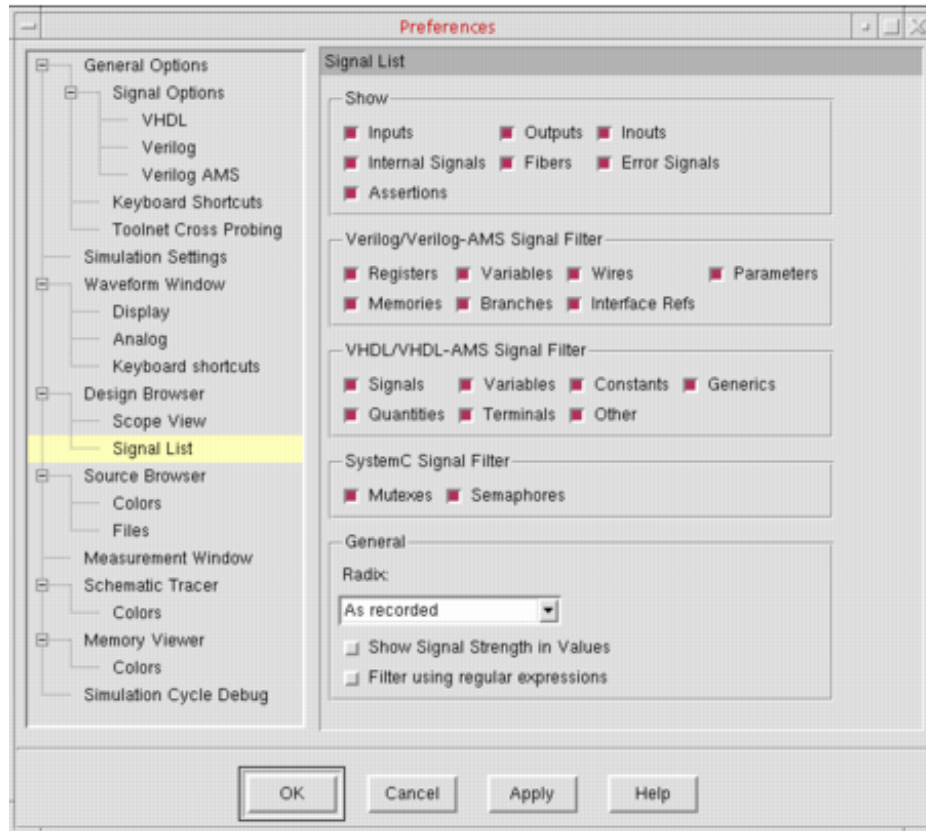
Setting Display Preferences for Verilog-AMS Objects

Items available on the Signal List pane for the AMS Designer simulator include analog branch objects.

To specify signal list preferences, do the following:

1. From the Design Browser, choose *Edit - Preferences*.
The Preferences form appears.
2. Under *Design Browser* (on the left side of the form), select *Signal List*.

The Signal List selections appear on the right side of the form.



3. On the right side of the form, select your preferences.

4. Click *Apply*.

The Design Browser modifies the Signal List pane to show the object types you select.

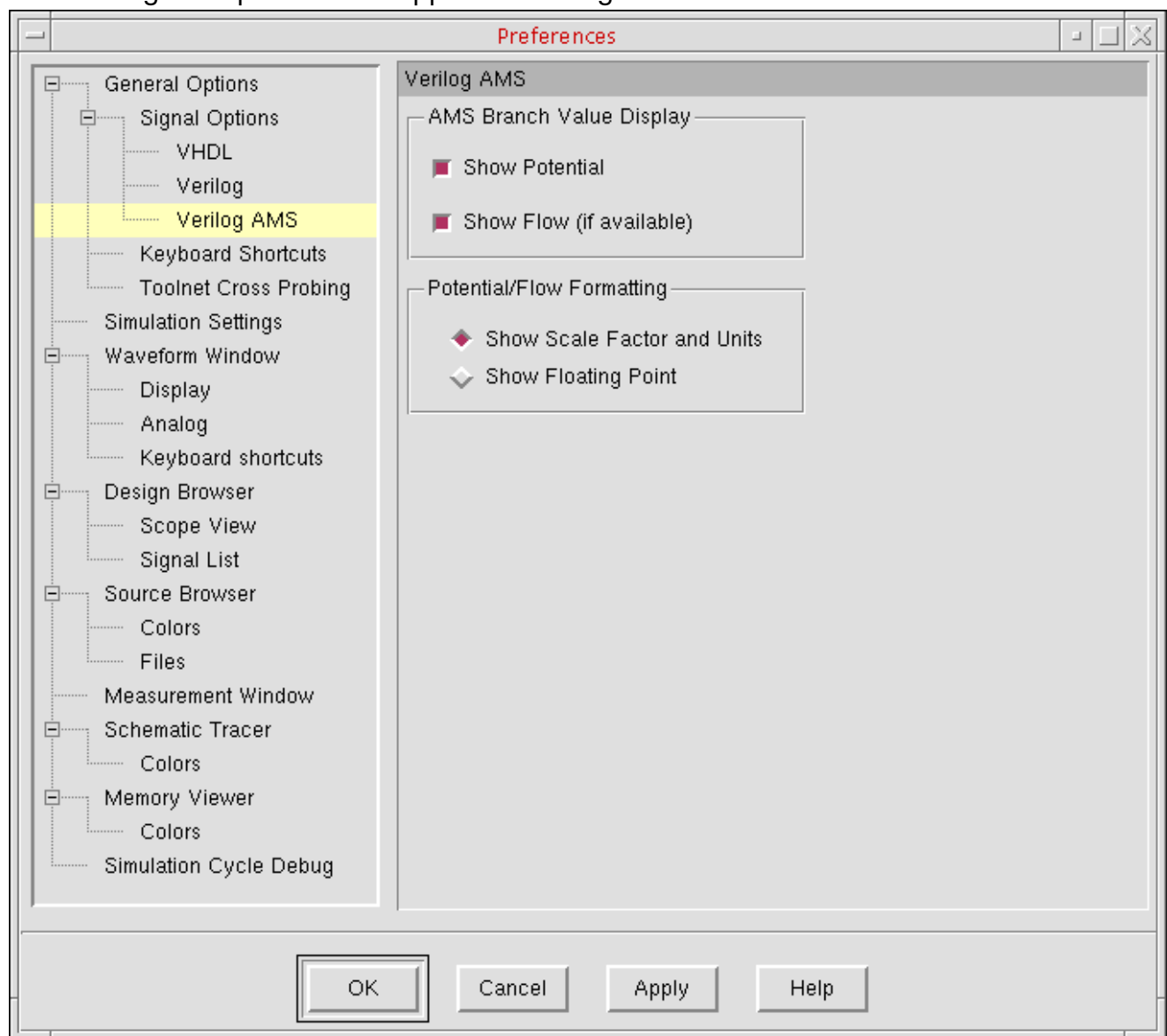
Related Topics

- [The Design Browser Window for AMS Designs](#)
- [Setting Formatting Preferences for Verilog-AMS Objects](#)
- [Using the Menus and Forms for AMS Designs](#)
- [Plotting Signals in the Waveform Window](#)

Setting Formatting Preferences for Verilog-AMS Objects

To specify formatting options for Verilog-AMS branches, do the following:

1. From the Design Browser, choose *Edit - Preferences*.
The Preferences form appears.
2. Under General Options (on the left side of the form), select *Signal Options - Verilog AMS*.
The Verilog AMS preferences appear on the right side of the form.



3. In the AMS Branch Value Display group box, select the options you want:

<i>Show Potential</i>	Show potential quantities, such as 5.44V
<i>Show Flow</i>	Show flow quantities, such as 3.45mA

4. In the Potential/Flow Formatting group box, select one of the following format options:

<i>Show Scale Factor and Units</i>	Use scalar multipliers and nature units; for example: <code>result=-30.646mV</code>
<i>Show Floating Point</i>	Use floating-point format; for example: <code>result=-0.0306461</code>

5. Click *Apply*.

Related Topics

- [Design Browser Window for AMS Designs](#)
- [SimVision Window Menu Bar](#)
- [Setting Display Preferences for Verilog-AMS Objects](#)
- [Plotting Signals in the Waveform Window](#)

Editing Source Information Using the Source Browser

1. Open the Source Browser.
2. In the Source Browser, navigate to the module of interest.
3. Click the *Edit Source* icon.



If the source file is a schematic, the Schematic Editing window displays the schematic view corresponding to the text view displayed in the Source Browser. If the source file is text, the text editor opens.

Related Topics

- [SimVision Window Menu Bar](#)
- [Using the Source Browser](#)
- [Editing Source Information](#)

Plotting Signals in the Waveform Window

You can use the SimVision Waveform window to display and analyze waveforms. How you plot a signal depends on the kind of probing that you use.

If you use Tcl probing for analog and digital objects,

1. Select the net in either the Schematic Editing window, or the Design Browser.
2. Choose *Windows - Waveform* in the Design Browser window to plot the signal in an existing Waveform window, or choose *Windows - New - Waveform* to plot the signal in a new Waveform window.

If you probe objects from the analog simulation control file,

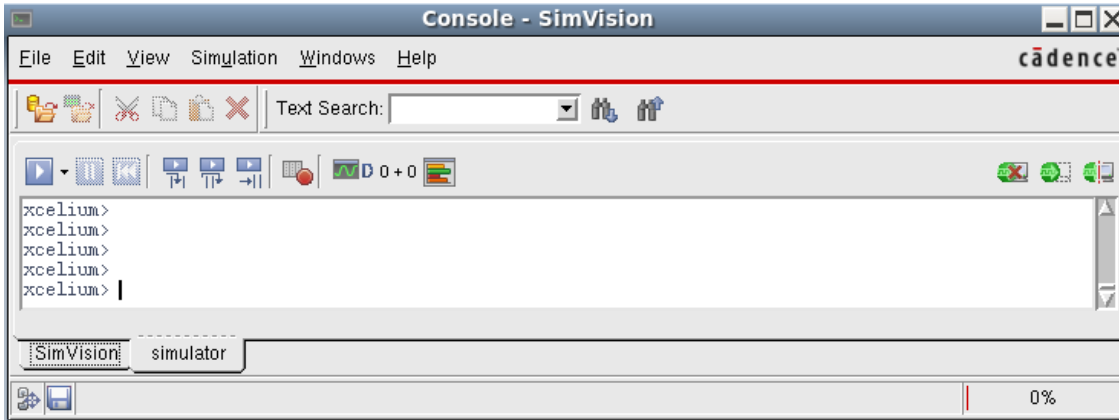
1. Start, but do not finish the simulation.
For example, you might choose *Simulation - Advance - Timepoint* in the Design Browser.
2. Select the net in either the Schematic Editing window or Design Browser.
3. Choose *Windows - Waveform* in the Design Browser window to plot the signal in an existing Waveform window, or choose *Windows - New - Waveform* to plot the signal in a new Waveform window.

Related Topics

- [Invoking SimVision](#)
- [Using the Menus and Forms for AMS Designs](#)
- [Plotting Signals in the Waveform Window](#)

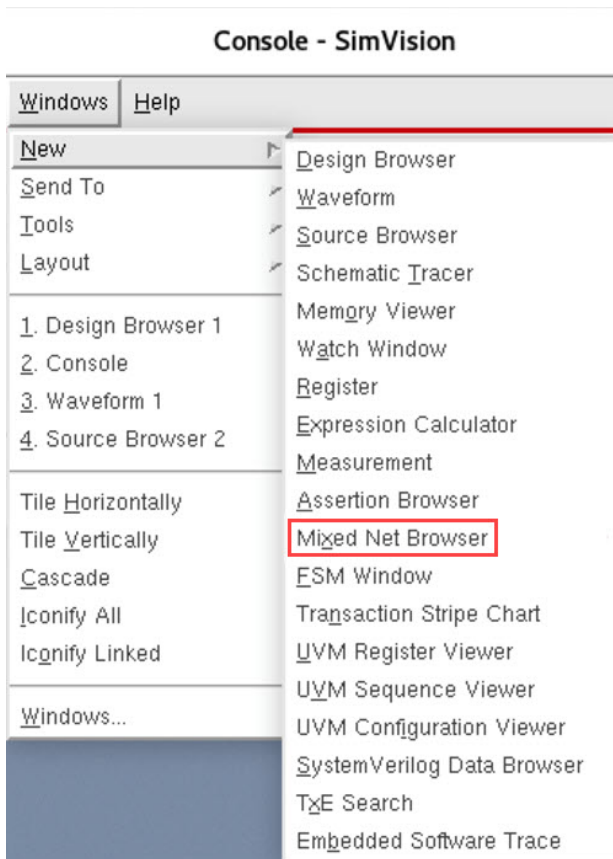
The Console Window

The Console window gives you access to the command languages for SimVisionMS and for the simulator you are running. Each tool is represented by a tab in the Console window. For example, if you run SimVisionMS with the simulator, the Console window contains a *SimVision* tab and a *simulator* tab, as shown in the figure.



You can type simulator, SimVision, or Tcl commands on the *SimVision* tab, and monitor the messages returned by the tool.

In SimVision MS, the *Windows - New* menu of the Console window has an additional option, *Mixed Net Browser*.



Related Topics

- [Invoking SimVision](#)
- [xrun Command-Line Options for Mixed-Signal Designs](#)

Cross-Probing Instances and Nets

Many of the applications that are part of the AMS Designer software support cross-probing. Cross-probing means selecting an instance or net in one application and having the same instance or net automatically selected in another application. For example, you can highlight an instance in the Cadence hierarchy editor (HED) and have the same instance appear highlighted in the schematic editing window. Similarly, you can highlight a net in the SimVision Design Browser and have the same net appear highlighted in the Waveform window.

See the following topics for more information:

- [Cross-Probing Instances](#)

- [Cross-Probing Nets](#)

Cross-Probing Instances

In AMS Designer, the applications that support cross-probing instances are SimVision, the Cadence [hierarchy editor](#), and the Cadence schematic editor. To enable cross-probing in the hierarchy editor and the schematic editor, do the following:

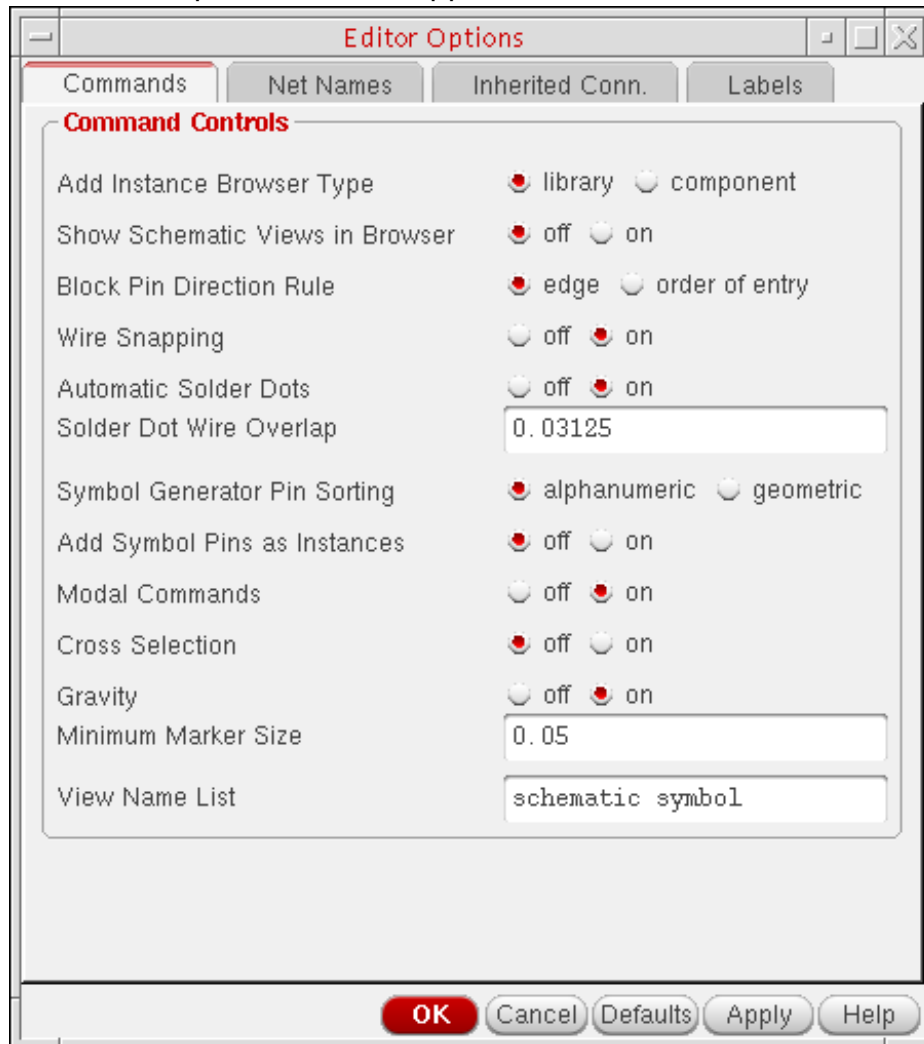
1. In the Cadence hierarchy editor window, choose *View - Tree*.

Instances in the design appear on the *Tree View* tab.

You can view cell bindings instead of instances by choosing *View - Parts Table*.

2. In the Virtuoso® Schematic Editor window, choose *Options - Editor*.

The Editor Options window appears.



1. Turn on *Cross Selection*.
2. Click *OK*.

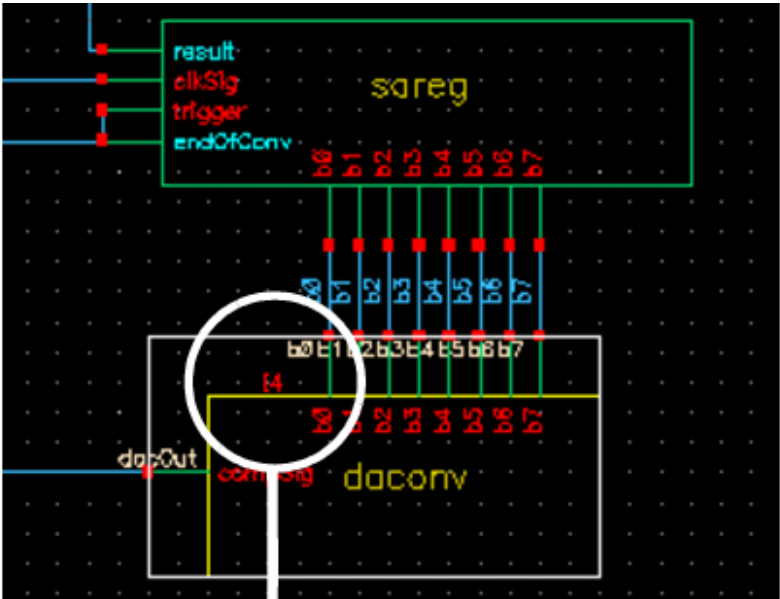
With these options set, you can cross-probe module instances as summarized in the following table. The first entry in each cell, marked with _p, refers to probing primitives. The second entry, marked with _m, refers to probing modules.

		Responding Application			
		Hierarchy Editor	Schematic Editor	SimVision	
Initiating Application	Hierarchy Editor		Yes _p Yes _m	No _p Yes _m	
	Schematic Editor	Yes _p Yes _m		No _p Yes _m	
	SimVision	No _p Yes _m	No _p Yes _m		

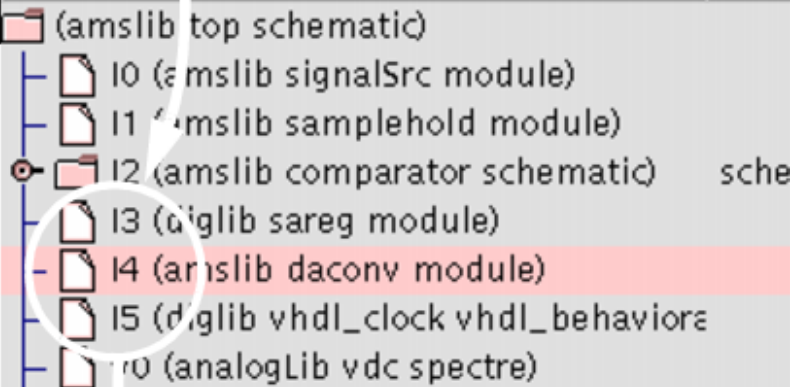
For example, if you select an instance in the schematic editor, the same instance is automatically selected in SimVision. Similarly, if you highlight a primitive in the hierarchy editor, the corresponding primitive in the schematic editor appears highlighted as well, and vice versa.

The following illustrates the cross-probing that occurs when the `daconv` component (instance `I4`) is selected in the Schematic Editing window.

Select instance I4 in the
Schematic Editing window.



Instance I4 highlights in the
hierarchy editor...



...and in the SimVision
windows.

```
18 sareg (* integer library_binding = "diglib"
19 .endOfConv( endOfConv ), .b5( b5 ), .b6(
20 .b0( b0 ), .clkSig( clkSig ), .b4( b4 ),
21 .result( compOut ), .trigger( endOfConv )
22
23 daconv #( refVolt(5.000000)) (* integer 1
24 *) I4 ( .b2( b2 ), .b5( b5 ), .b6( b6 ),
25 .compSig( dacOut ), .b7( b7 ), .b0( b0 ),
```

Cross-Probing Nets

In AMS Designer, the applications that support cross-probing nets are SimVision, the schematic editor, and the Waveform window. You can enable cross-probing in the schematic editor by following steps 2 through 4 in "[Cross-Probing Instances](#)". With these options set, you can cross-probe nets as summarized in the following table.

Initiating Application	Responding Application			
		Schematic Editor	SimVision	Waveform
	Schematic Editor		Yes	Yes
	SimVision	Yes		Yes
	Waveform	No	Yes	

For example, if you select a net in the schematic editor, the same net automatically highlights in the SimVision window. However, if you select a net in the Waveform window, the schematic editor window does not reflect the selection.

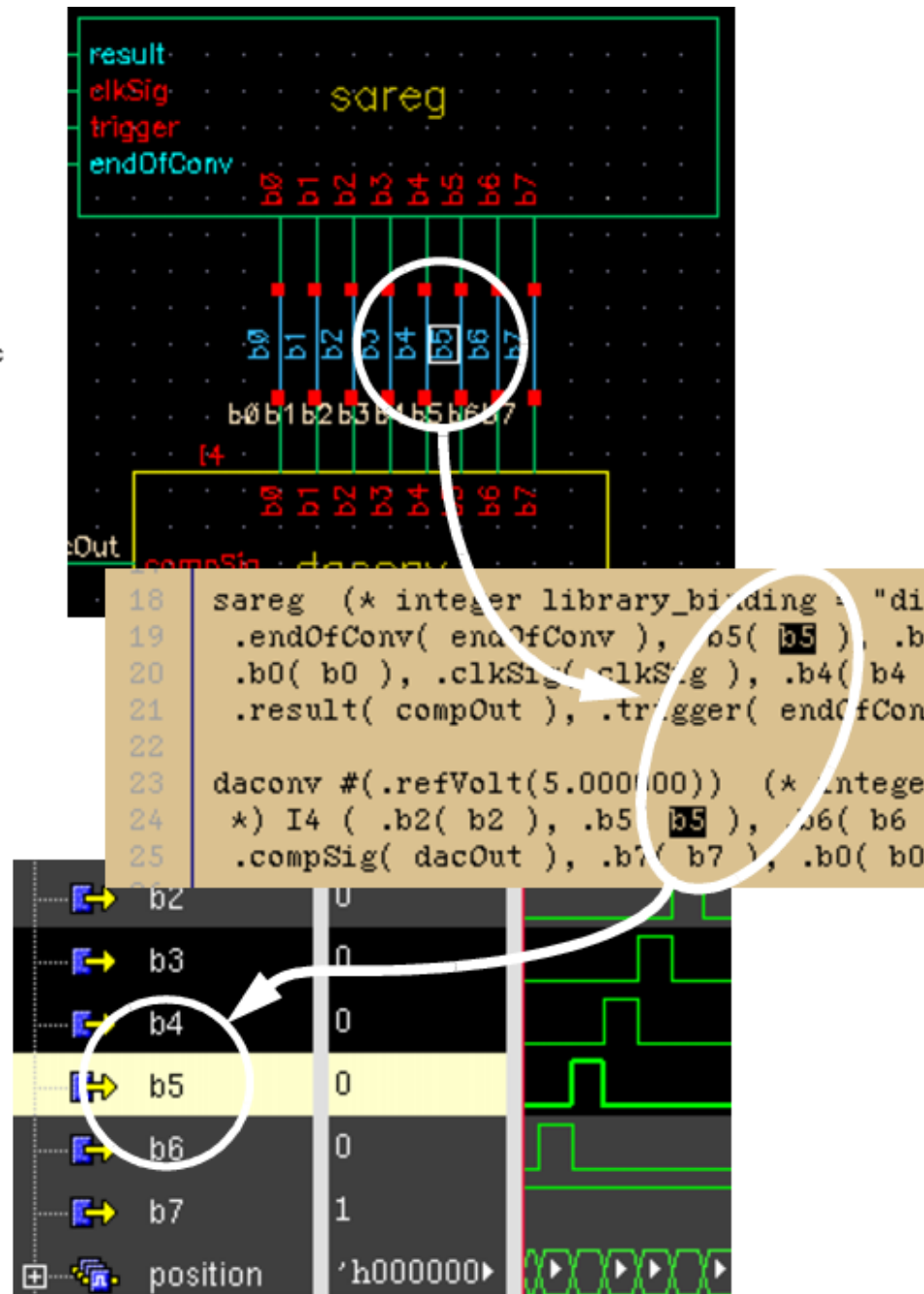
You cannot cross-probe global signals with AMS Designer.

The following illustrates the cross-probing that occurs when the `b5` net is selected in the Schematic Editing window.

Select net b5 in the Schematic Editing window.

Net b5 highlights in the SimVision window...

...and in the Waveform window.



Related Topics

- [Using the Menus and Forms for AMS Designs](#)
- [Using the Source Browser](#)
- [Editing Source Information](#)

- [Plotting Signals in the Waveform Window](#)

Analog Mnemonic Maps in SimVision

Mnemonic maps associate signal values with strings, colors, line shapes, and icons. For example, you can define a mnemonic map to associate a name with each state of a state machine or a mnemonic map to display all x values with a background color of your choice.

When you create an analog mnemonic map, you define not only the text that is displayed for a signal value, but also the background color of the waveform in the SimVision Waveform window.

You can create analog mnemonic maps in one of the following methods:

- **Use the SimVision TCL command.** You must use the `-analog` option with the `mmap new` SimVision command. The syntax is:

```
mmap new -analog -name label -contents {  
{- range { <lowValue> <highValue>} -label "name" -bgcolor}
```

You can define one or more ranges for a given mnemonic map entry using `-range {<lowValue> <highValue>}`. Each entry is specified as a list with a range, a label, and background color that specifies how a waveform is drawn when its value matches one of the value ranges for the mnemonic map entry.

So, for a signal set with a map with value in one of the value ranges, a box with a label is displayed in the Waveform trace.

For more information on the `mmap` command, refer to the *SimVision Tcl Commands Guide*.

- **Define analog mnemonic maps in a script file and source it to SimVision.** In the SimVision GUI, choose *File – Source Command Script*; and in the *Select SimVision Command Script* dialog box select the command file with Analog Mnemonic Maps.

Analog Mnemonic Map Application

To apply a mnemonic map to a signal, perform one of the following methods:

- In the SimVision Waveform window, then click *Format – Radix/Mnemonic* and choose the analog mnemonic map.
- In the SimVision Waveform window, right-click the signal value in the *Cursor* column and choose *Radix/Mnemonic – analog mnemonic map*.

Limitations

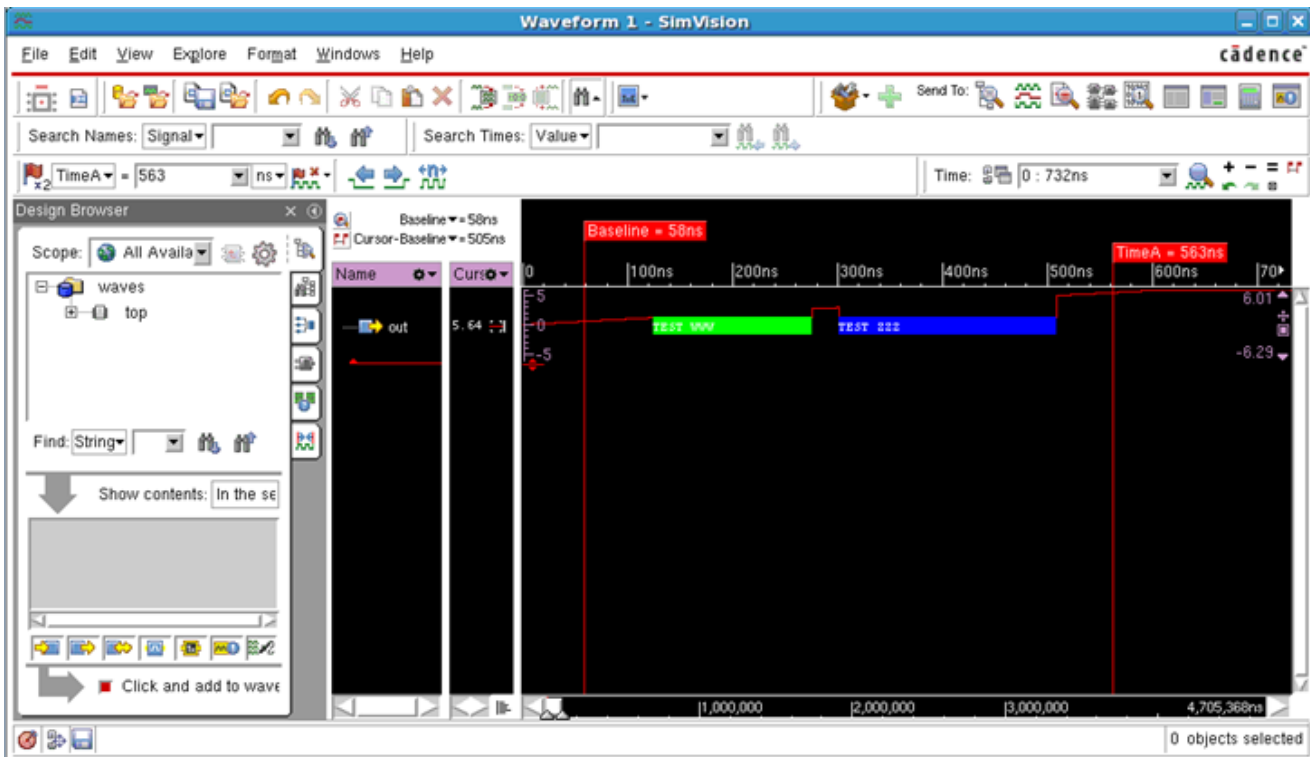
The following limitations exist with the analog mnemonic map feature:

- Analog mnemonic maps are not listed in the *Mnemonic Maps* tab of the *Properties* window.
- Analog mnemonic maps cannot be applied to non-analog traces such as integers.
- Analog Mnemonic Maps cannot be applied on composite types such as SV UDN wreal/vams, wreal/var real, and SV UDT/UDN structures. You can only select members of a composite type (individually or as a group) and then apply the analog mnemonic map to those selected signal(s).
- You cannot apply an analog mnemonic map to a group of signals from a SV UDT/UDN structure, if one or more of the signals in that group are of a non- analog type (logic type or integer type).
- Only the `-label`, `-bgcolor` options of the `mmap` Tcl command are supported. The other `mmap` options documented in the *SimVision Tcl Commands Guide* are not supported.

Example

The following example illustrates using the `mmap` TCL command. It creates an analog mnemonic map, `Analog Mnemonic Map` and creates two entries for the mnemonic map. Each entry is specified with a range, a label, and the background color for the mnemonic map.

```
mmap new -analog -name "Analog Mnemonic Map" -contents {  
{-range {1.25 2.76} -label "TEST WWW" -bgcolor green}  
{-range {3.01 5.09} -label "TEST ZZZ" -bgcolor blue}
```

Related Topics

- [Invoking SimVision](#)
- [Using the Menus and Forms for AMS Designs](#)
- [Plotting Signals in the Waveform Window](#)

Real Value Probe Filtering

The AMS Designer simulator allows you to specify a precision (delta value) for real number probes to remove recorded values whose delta is less than the precision specified. This helps reduce the waveform database size for real number model simulations, improve simulation performance; and maintain the simulation model accuracy.

You can enable real value filtering for all real value probes using the `-shm_filter_group <precision>` global option, with `xrun` or `xmsim` commands.

When the `-shm_filter_group` option is used, the delta from the previously dumped value is computed. If this delta is at or greater than the precision tolerance, then the value is dumped to the waveform database, otherwise, it is skipped. This reduces the waveform database size by not recording values less than the specified precision.

The value for `precision` can be integer, float or scientific notation. When an integer value is specified it represents the number of points after the decimal place; so, an integer value of 2 would represent a delta value of 0.01. When float or scientific notation are used its value is used as the delta value.

This option does not filter probe values computed by the analog solver (electrical signals).

Debugging Mixed-Signal Designs

This topic explains how to debug mixed-signal designs.

To know more, see the following topics:

- [Displaying Debug Settings](#)
- [Mixed Network Debugging](#)
- [Managing Breakpoints](#)
- [Managing Databases](#)

Terminology

The descriptions in this chapter use terminology that might be new to you. The terms have to do with the way that the compiler simulates mixed analog and digital designs.

Term	Definition
analog solver	The part of the AMS Designer simulator that simulates the analog portions of a design. Some SimVision capabilities appear only when the analog solver is active.
digital solver	The part of the AMS Designer simulator that simulates the digital portions of a design. Some capabilities are enabled only when the digital solver is active.
analog context	The context of statements that appear in the body of an <code>analog</code> block.
digital context	The context of statements that appear in a location other than an <code>analog</code> block.

Downgrading Mixed-Signal Initialization Errors

You can downgrade the mixed-signal simulation and elaboration initializations errors to warnings to allow faster debugging and resolution of issues. To downgrade errors to a warning, use the `xrun/xmelab` command-line option, `-dms_fast_debug`. The tool downgrades the EXNEUS and EXANCU errors to warnings and completes simulation and elaboration without any interruption.

The snapshot of simulation or elaboration, where error messages are downgraded to a warning, is not simulatable.

Forcing and Releasing Signal Values

You can ask "what if" questions about your model by interactively forcing objects to desired values and seeing if the patch fixes the problem. If it does, you can then edit your source file to incorporate the change.

The object that is being forced must have write access. To specify write access, use the `-access` or `-afile` option when you elaborate the design with `xmelab`.

To set an object to a given value,

- If you are using the Tcl command-line interface, use the `force` command to set a specified object to a given value and force it to retain that value until you release it with a release command or place another force on it.

You cannot use the `force` command on an analog object. In addition, you cannot use the `force` command on digital objects while the analog solver is active.

- If you are using SimVision, choose *Simulation - Create Force* to force a signal to a given value. To release a signal, position the pointer over the signal, click the right mouse button, and select *Release Force* to release the signal. This choice is not active while the analog solver is active because you cannot use the `force` command in that circumstance.

You cannot use the `force` command on an analog object. In addition, you cannot use the `force` command on digital objects while the analog solver is active.

Related Topics

- `force`
- `release`

Managing Breakpoints

After setting breakpoints, you can display information on breakpoints, disable breakpoints, enable previously disabled breakpoints, and delete breakpoints.

If you are using the command-line interface:

- Use the `stop` command with the `-show`, `-disable`, `-enable`, or `-delete` modifier. The argument for these modifiers can be:
 - A break name or a list of break names
 - A pattern
 - The asterisk (*), which matches any number of characters
 - The question mark (?), which matches any one character
 - `[characters]`, which matches any one of the characters
 - Any combination of literal break names and patterns

If you are using SimVision,

- Choose *Simulation - Show - Breakpoints* and use the *simulator:Breakpoints* settings in the Properties window

Managing Databases

You can open, close, disable, enable, and display information about databases. See the following topics for more information:

- [Opening a Database](#)
- [Displaying Information about Databases](#)
- [Disabling a Database](#)
- [Enabling a Database](#)
- [Closing a Database](#)

Opening a Database

You can open two types of databases: SHM or VCD. SHM databases support probing both analog and digital signals. VCD databases support probing only digital signals.

When you open a VCD database, the software converts it to SHM format. You can export a database to VCD format.

- If you are using the Tcl command-line interface, type `database -open` to open either type of database.

Partial syntax:

```
database [-open] dbase_name [-shm | -vcd] [-into file_name ][-default]
```

See "Database" in the "Using the Tcl Command-Line Interface" chapter, of *Cadence Verilog Simulation User Guide* for complete syntax and details on the `database` command.

- If you are using the Cadence® SimVision environment, choose *File - Open Database* and fill in the Open Database form.
See "Managing Simulation Databases" in the *SimVision User Guide* for information about managing your databases.

You can also open a database from a digital context with the `$shm_open` system task in your Verilog®-AMS code. A database opened in this way does not support probing analog signals.

Displaying Information about Databases

- If you are using the Tcl command-line interface, type `database -show` to display information about databases.

Syntax:

```
database -show [{ dbase_name | pattern } ...]
```

- If you are using the Cadence SimVision environment, choose *Simulation - Show - Databases*.

Disabling a Database

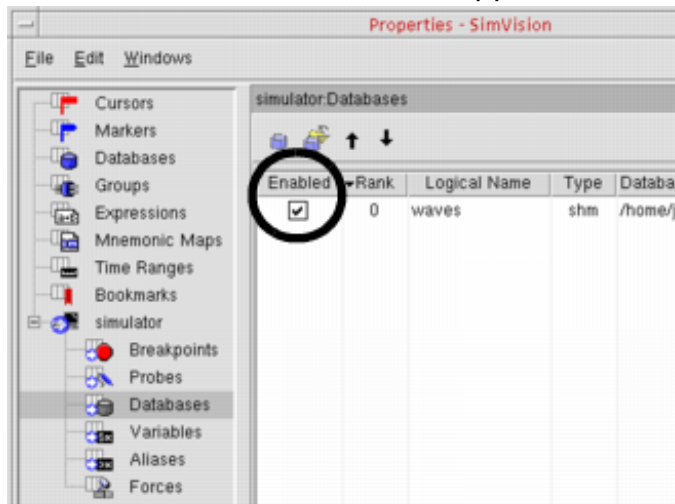
- If you are using the command-line interface, type `database -disable` to disable either type of database temporarily.

Syntax:

```
database -disable { dbase_name | pattern } ...
```

- If you are using SimVision, do the following:
 - a. Choose *Simulation - Show - Databases*.

The simulator:Databases view appears in the Properties window.



- b. Remove the mark from the *Enabled* check box for the database you want to disable.

Enabling a Database

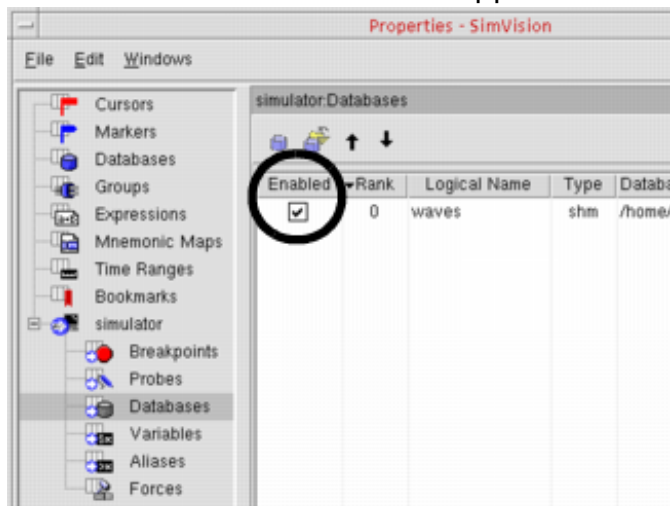
- If you are using the command-line interface, type `database -enable` to enable a previously disabled database.

Syntax:

```
database -enable { dbase_name | pattern } ...
```

- If you are using SimVision, do the following:
 - a. Choose *Simulation - Show - Databases*.

The simulator:Databases view appears in the Properties window.



- b. Mark the *Enabled* check box for the database you want to enable.

Closing a Database

- If you are using the Tcl command-line interface, type `database -close` to close either type of database.

Syntax:

```
database -close { dbase_name | pattern } ...
```

- If you are using SimVision, do the following:
 - a. Choose *File - Close Database/Simulation* to close a simulation database.
 - b. On the Close Database/Simulator form, select the database you want to close.
 - c. Click *OK*.

Mixed Network Debugging

For mixed network analysis and debugging, you can use the `-dms_mxnet_sql` option. This option creates the mixed network database and generates a *mxnet_sql.db* file, which is an SQL representation of the mixed network database.

You can specify this option to debug information about connect modules, using standard SQL browsers. The information is generated at the start of the simulation. So, you can use this option instead of `-ieinfo` to access similar information about connect modules within the context of a mixed network. In larger designs, the `-dms_mxnet_sql` option is a faster and more efficient way to dump a searchable database of the connect modules and their hierarchical connectivity details in mixed-signal designs.

Model Hierarchy for Spectre AMS Designer

The Spectre AMS Designer Simulator supports hierarchical designs by allowing you to embed models in other models. Levels of hierarchy in a design are called scopes. To create a scope, you nest objects within design units by instantiating them. Instantiation allows one design unit to incorporate a copy of another into itself.

Each scope in a design hierarchy has a unique hierarchical path. For Verilog-AMS, elements in the path are separated by a period (`.`). You can specify paths

- Fully from the top level of the hierarchy

Full paths begin with the name of a Verilog-AMS top-level module. For example:

```
top.board.counter  
top.vending.drinks.count_cans.in1
```

- Relative to the current debug scope

For example, if the current debug scope is `top.board`, the path `counter` refers to a scope within the scope `board`, which is within the top-level module `top`.

You traverse the model hierarchy by setting the scope to an instantiated object. If you are using the Tcl command-line interface, use the `scope -set` command. For example, if the current debug scope is the top level, and you want to move down one level to a scope called `board`, use the following command:

```
scope -set board
```

If you are at the top level and want to move down to a scope within `board` called `counter`, use the

following command:

```
scope -set board.counter
```

You can specify a full path from any debug scope. For example, if the current scope is `board:counter`, you can move up to the top level (module `top`) with the following command:

```
scope -set top
```

The `scope` command has several options besides `-set`. These options let you

- Describe the items declared within a scope (`-describe`)
- Display the drivers of the objects declared within a scope (`-drivers`)
- Print the source code, or part of the source code, for a scope (`-list`)
- Display scope information (`-show`)

Related Topic

- [scope](#)

Paths and Mixed-Language Designs

In VHDL, you use a colon to separate path elements. A full path begins with a colon, which represents the top-level design unit. The first path element is an item in the top-level scope. The following are examples of fully specified paths:

```
:vending
```

```
:vending:drinks
```

```
:vending:drinks:sig2
```

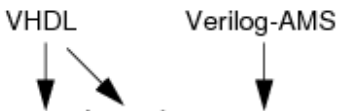
Relative paths do not begin with a colon. For example, if the current debug scope is `:vending`, the path `drinks` refers to a scope within the scope `vending`, which is within the top-level design unit.

In a mixed-language simulation, you can use a period or a colon as the path element separator. The Spectre AMS Designer Simulator uses the following rules:

- If the path begins with a colon, the path is a full path starting at the VHDL top-level scope. A colon by itself refers to this scope. You cannot use any other special character at the start of a path.
- If the path does not start with a colon, and the first path element is in the current debug scope,

the path is relative to the debug scope. If the first path element is not in the current debug scope, the simulator assumes that the path is a full path whose first path element is the name of one of the top-level Verilog-AMS modules.

For example, suppose that you have a mixed Verilog-AMS and VHDL design, where the top-level design unit is VHDL. With Tcl commands, you can use both path element separators interchangeably (except at the beginning of a path, as specified above), as shown in the following examples:



```
xcelium> scope -set :board:counter:a
xcclium> scope -set board:counter.a
xcclium> scope -set board.counter:a
xcclium> scope -set board.counter.a
```

Because VHDL is case insensitive (except for escaped names) and Verilog-AMS is case sensitive, each element of a mixed language path is either case sensitive or case insensitive, depending on its language context. When the parser looks for a name in a Verilog-AMS scope, it is case sensitive; when it looks for a name in a VHDL scope, it is case insensitive.

The syntax that you use for name expressions is also interchangeable. Name expressions are bit-selects, part-selects, and array element specifiers in Verilog-AMS, and array element and record field specifiers in VHDL. Index specifiers are also used in VHDL scope names when the scope is created by a for-generate statement.

Verilog-AMS index specifiers use square brackets, and a colon separates the left and right bounds of the range (for example `[7:0]`). VHDL index specifiers use parentheses, and the keyword `TO` or `DOWNTO` separates the left and right bounds of the range (for example, `(7 downto 0)`).

You can use either style with VHDL index ranges. Using a colon in a VHDL index range is the same as using the direction with which that index range was declared.

Record field specifiers apply only to VHDL objects. Use a period to separate the object name from the record field.

The following pairs of Tcl commands are identical.

```
xcelium> scope foo_array(2)xcclium> scope foo_array[2]

xcclium> value sig[7:0]xcclium> value sig(7:0)

xcclium> value sig[7]xcclium> value sig(7)

xcclium> describe sig[7 downto 0]xcclium> describe "sig(7 downto 0)"
```

You can use either Verilog-AMS or VHDL escaped name syntax in paths. For Verilog-AMS, escaped names begin with a backslash and are terminated with a space. For example:

```
abc.xyz.\some_name .signal
```



↑
space

For VHDL, escaped names begin and end with a backslash (for example, `\w3.OUT\`).

The following two `value` commands are identical:

```
xcelium> value top.vending.#{@w3.OUT }
```

```
xcelium> value top.vending.#{@w3.OUT\}
```

Setting and Deleting Probes

You save the values of objects to a database by probing them. You can view these database values using SimVision.

- If you are using the Tcl command-line interface, use the `probe` command to set, disable, enable, delete, and display information about probes for analog and digital objects.
- If you are using SimVision, choose Simulation - Create Probe to set a probe. To disable, enable, or delete a probe, choose *Simulation - Show - Probes* and use the *simulator:Probes* settings in the Properties window.

For Verilog-AMS, digital objects can be probed only if they have read access.

See the following topics for Tcl command syntax:

- [Setting a Probe Using the Tcl probe Command](#)
- [Displaying Information about Probes Using the Tcl probe Command](#)
- [Disabling a Probe Using the Tcl probe Command](#)
- [Enabling a Probe Using the Tcl probe Command](#)
- [Deleting a Probe Using the Tcl probe Command](#)

Setting a Probe Using the Tcl probe Command

If you are using the Tcl command-line interface, type `probe -create`.

Partial syntax:

```
probe [-create] [{ object | scope_name }...]
      [-shm | -vcd | -database dbase_name ]
      [-all]
      [-nettype nettype]
      [-depth { n | all | to_cells}]
      [-inputs]
      [-name probe_name ]
      [-outputs]
      [-ports]
      [-screen [-format format_string ] [-redirect filename ]
              objects ]
      [-variables]
```

If you do not specify an `object` or `scope_name` argument, you must use one of the following options: `-inputs`, `-outputs`, `-ports`, or `-all` to specify which objects to probe.

The `-all` option probes all declared objects within a scopes, except for VHDL variables. To include VHDL variables in the probe, include the `-variables` option (`-all -variables`).

Displaying Information about Probes Using the Tcl `probe` Command

If you are using the Tcl command-line interface, type `probe -show` to display information about the probes you have set.

Syntax:

```
probe -show [{ probe_name | pattern } ...]
```

Disabling a Probe Using the Tcl `probe` Command

If you are using the Tcl command-line interface, type `probe -disable` to stop a probe temporarily.

Syntax:

```
probe -disable { probe_name | pattern } ...
```

Enabling a Probe Using the Tcl `probe` Command

If you are using the Tcl command-line interface, type `probe -enable` to enable a probe that you disabled previously.

Syntax:

```
probe -enable { probe_name | pattern } ...
```

Deleting a Probe Using the Tcl probe Command

If you are using the Tcl command-line interface, type `probe -delete` to delete a probe.

Syntax:

```
probe -delete { probe_name | pattern } ...
```

Related Topic

- [probe](#)

Stepping through Lines of Code

You can examine the order in which the simulator executes the statements in your model by stepping through the simulation line by line.

This capability is not supported for structural Verilog-AMS code.

While you can always single-step through the statements in the `analog` block of a module, outside the `analog` block, you cannot single-step or set line breakpoints in a particular design unit unless you compiled the unit with the `-linedebug` option. If you compiled the unit without this option, you can use the `run -step` or `run -next` command to run the simulation until the next point where it can stop. If execution passes to a unit that you compiled with `-linedebug`, full single-stepping resumes.

- If you are using the Tcl command-line interface:
 - Use `run -step` to simulate to the next executable line of code in any scope. This command runs one statement, stepping into subprogram calls.

The `-step` option does not step into function calls made by an analog statement. In this situation, the behavior of the `-step` option is identical to the behavior of the `-next` option.

- Use `run -next` to run one statement, stepping over any subprogram calls.

- If you are using SimVision, choose *Simulation - Step* or *Simulation - Next*.

Related Topic

- [run](#)

Breakpoints

You can interrupt the simulation by setting breakpoints. The type of breakpoints you can set depend on the language you are using as follows:

Breakpoint Type	Verilog-AMS	VHDL
Condition breakpoints	Yes	Yes
Line breakpoints	Yes	Yes
Signal breakpoints	Yes	Yes
Time breakpoints	Yes	Yes
Subprogram breakpoints	Yes	Yes
Process breakpoints	No	Yes

Setting a Condition Breakpoint

A condition breakpoint stops the simulation when a specified condition is true. This type of breakpoint is particularly useful when you want to stop the simulation at the instant a signal has an incorrect value.

A condition breakpoint triggers when any digital object referenced in the conditional expression changes value (wires, signals, registers, and variables) or is written to (memories) and the expression evaluates to true (nonzero). Condition breakpoints are not triggered by changes in analog objects, but you can include analog objects in the conditional expression and their values are used when the condition is evaluated (due to a digital object changing value).

To set a condition breakpoint,

- If you are using the Tcl command-line interface, type `stop -condition`.
- If you are using SimVision, choose *Simulation - Set Breakpoint - Condition*.

A condition breakpoint takes a Tcl expression as an argument.

The simulator does not support breakpoints on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires.

For Verilog-AMS, objects included in a conditional expression must have read access. An error is printed if the object does not have read access.

Setting a Line Breakpoint

A line breakpoint stops the simulation at a specified line in the source code. You set this type of breakpoint when you want to simulate to a certain point and then single-step through lines of code.

You cannot set a line breakpoint unless you have compiled with the `-linedebug` option.

You can set a line breakpoint only in pure digital code when you are using AMS Designer simulator with the Spectre solver and the simulation front end (SFE) parser.

To set a line breakpoint,

- If you are using the Tcl command-line interface, type `stop -line` option.
- If you are using SimVision, choose [Simulation - Set Breakpoint - Line](#).

Setting a Signal Breakpoint

A signal breakpoint stops the simulation when a specified wire or signal changes value or when the simulation writes a value to a register, memory, or variable. Use a signal breakpoint when you want the simulation to stop every time a signal changes value or when you want to see the value of signals when some condition is true (for example, on every positive edge of the clock).

To set a signal breakpoint,

- If you are using the Tcl command-line interface, type `stop -object`.
- If you are using SimVision, choose [Simulation - Set Breakpoint - Signal](#).

For Verilog-AMS, the object specified as the argument must have read access for the breakpoint to be created.

The simulator does not support breakpoints on analog objects (nets, branches, or variables). Nor does the simulator support breakpoints on individual bits of registers or variables. For example, the following command generates an error message:

```
xcelium> stop -create -object data[1]
```


Setting a Time Breakpoint

A time breakpoint stops the simulation at a specified time. The time can be absolute or relative (the default). Absolute time breakpoints are automatically deleted after they trigger. Relative time breakpoints are periodic, stopping, for example, every 10 ns.

This type of breakpoint is usually set when you want to advance the simulation to a certain time point before beginning to debug or when you want to stop the simulation at regular intervals to examine signal values.

To set a time breakpoint,

- If you are using the Tcl command-line interface, type `stop -time`.
- If you are using SimVision, choose [Simulation - Set Breakpoint - Time](#).

Setting a Process Breakpoint

For VHDL, a process breakpoint stops the simulation when a named process starts executing or resumes executing after a wait statement.

You must compile with the `-linedebug` option to enable the setting of source line and process breakpoints.

To set a process breakpoint,

- If you are using the Tcl command-line interface, type `stop -process`.
- If you are using SimVision, choose [Simulation - Set Breakpoint - Process](#).

Setting a Subprogram Breakpoint

For VHDL or Verilog, a subprogram breakpoint stops the simulation when the simulation reaches a VHDL or Verilog function or procedure. You can then use the `step` command to step into the function or procedure to view values of objects.

To set a subprogram breakpoint,

- If you are using the Tcl command-line interface, type `stop` command with the `-delta` option.
- If you are using SimVision, choose [Simulation - Set Breakpoint - Subprogram](#).

Related Topics

- [Access to Digital Simulation Objects](#)
- [Managing Breakpoints](#)
- [stop](#)

Depositing Values to Signals

You can ask "what if" questions about your model as you debug by interactively depositing a value to a specified object.

When you deposit a value to an object, behaviors that are sensitive to value changes on the object run when the simulation resumes, just as if the value change was caused by the Verilog-AMS or VHDL code.

You can deposit a value to an object immediately, at a specified time in the future, or after a specified delay. You can also specify that you want to deposit the value after an inertial delay or after a transport delay. A deposit without a delay is similar to a force in that the specified value takes effect and propagates immediately. However, it differs from a force in that future transactions on the signal are not blocked.

For VHDL, you can deposit to ports, signals, and variables if no delay is specified. If a delay is specified, you cannot deposit to variables or to signals with multiple sources.

For Verilog-AMS, you can deposit to ports, signals (wires and registers), and variables.

For Verilog-AMS, the object that you want to deposit a value to must have write access. An error is printed if it does not. To specify write access, use the `-access` or `-afile` option when you elaborate the design with `xmelab`.

You cannot use the `deposit` command on an analog object. In addition, you cannot use the `deposit` command on digital objects while the analog solver is active.

To deposit a value to an object,

- If you are using the Tcl command-line interface, type `deposit`.
- If you are using SimVision, choose [Simulation - Deposit Value](#).

Displaying Information about Simulation Objects

To display information about a simulation object, including its declaration,

- If you are using the Tcl command-line interface, type `describe`.
- If you are using SimVision, right-click the object and select *Describe*.

Displaying the Drivers of Signals

You can display a list of all of the contributors to the value of a specified digital object.

You cannot list drivers for analog variables, analog nets, or branches.

To display the driver of signals,

- If you are using the Tcl command-line interface, type `drivers`. For example:

```
xcelium> drivers board.count
```

You can use the `scope -drivers [scope_name]` command to display the drivers of each object declared in a specified scope. You can use the `scope -describe [scope_name]` command to give the description of each object that is declared within the specified scope.

For Verilog-AMS, the `drivers` command cannot find the drivers of a wire or register unless the object has connectivity access. However, even if you have specified access to the object, its drivers might have been collapsed, combined, or optimized away. In this case, the output of the `drivers` (or `Show - Drivers`) command might indicate that the object has no drivers.

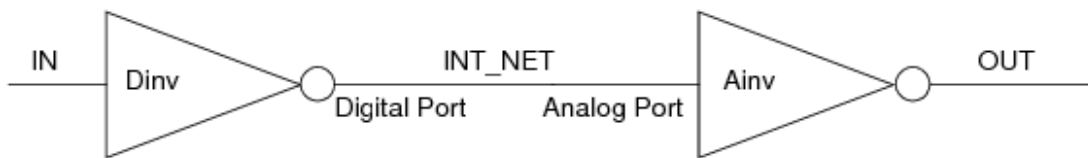
Related Topics

- [describe](#)
- [drivers](#)
- [scope](#)

Automatically-Inserted Connect Modules

In this release, automatically-inserted connect modules are not visible in SimVision. As a consequence, the values returned by probing might not be what you expect. Connect modules are used when connected ports have disciplines of different domains (such as logic and electrical) so you need to be aware of the effects of connect modules that are automatically inserted, but not visible, between such ports. Note that manually inserted connect modules are visible in SimVision.

To better understand, consider the following example where the top module contains two inverters. The ports of the `Dinv` inverter are of the logic discipline and the ports of the `Ainv` inverter are electrical.



At time t , the value of `IN` is 0. Because there are two inverters, the value of `INT_NET` is 1 and the value of `OUT` is 0.

If you change scope into the `Dinv` module and probe the value of the digital port, you see that the value is 1. However, if you trace the signal to `INT_NET` in the top module, you find that the value depends on the discipline of that net. If `INT_NET` is of the logic discipline, the value is 1 as expected. But if `INT_NET` is of the electrical discipline, the value is a real number calculated by the analog solver. In a typical case, the value might be between 2.5 V and 5 V.

Similarly, if you trace the signal into the analog port of `Ainv`, where, again, the disciplines of the driver and receiver do not match, a similar issue arises.

Waveforms

You can use the SimVision Waveform window to display and analyze waveforms.

This section explains how to open a database, how to probe signals, and how to open the [SimVision Waveform window](#).

The objects that you want to probe to a database must have read access. Analog objects have read access, but, by default, Verilog-AMS digital objects in the design do not have read access. Use the `-access +r` option or the `-afile access_file` option when you elaborate the design to provide read access.

Creating a Database and Probing Signals

You can open a database, probe signals, and save the results in the database by typing Tcl commands at the prompt or by using the graphical user interface.

You also can use a set of system tasks from the digital context to open an SHM database, probe signals, and save the results. You must type these system tasks in the Verilog-AMS code prior to simulation. The system tasks are

System task	Description
<code>\$shm_open();</code>	Opens a simulation database.
<code>\$shm_probe();</code>	Selects signals whose simulation value changes will enter the simulation database.
<code>\$shm_close;</code>	Closes a simulation database.

Example:

```
initial
begin
    $shm_open("waves.shm");
    $shm_probe();
    #1 $stop; // stop simulation at time 1
end
```

When you send signals to the Waveform window and click *Run*, SimVision creates a probe for every signal in the Waveform window. In this way, you can quickly and easily probe objects.

Opening a Database with \$shm_open

Use the `$shm_open` system task from a digital context to open an [SHM](#) database. You cannot use a database created in this way to save data for analog objects.

Syntax:

```
$shm_open ( [ " db_name " ], [ is_sequence_time ], [ db_size ], [ is_compression ] );
```

Where,

"db_name" specifies the filename of the simulation database. If you do not specify the database name, a database called `waves.shm` is opened in the current directory.

`is_sequence_time` dumps all value changes to the database.

By default, when probing to an SHM database, the simulator discards multiple value changes for an object during one simulation time and dumps only the final value at the end of that simulation time. Specify 1 for the `is_sequence_time` argument if you want to dump all value changes to the SHM database. For example,

```
$shm_open("mywaves.shm", 1, , ) ;
```

`db_size` Specifies the maximum size (in bytes) of the transition file (`.trn` file).

The simulator maintains the size limit by discarding the earliest recorded values as new values are dumped, such that the database always contains the most recent values for each probed object.

When the size limit is exceeded, the waveform window displays an unknown value for each object from the beginning of the simulation to the time of the first non-discarded value.

The SHM database uses about 2.5Mb of disk space, even if you specify a lower limit. However, the database size will not exceed the limit if the limit is greater than 2.5Mb. For example,

```
$shm_open("mywaves.shm", 1, 250000,);
```

`is_compression` Compresses the SHM database to reduce its size. The default setting is 0. Specify 1 to compress the database file. For example,

```
$shm_open("mywaves.shm", 1, , 1);
```

Probing Signals with `$shm_probe`

The syntax for the `$shm_probe` system task, which can be used only on digital objects and must be used from a digital context, is

```
$shm_probe( scope0 , node0 , scope1 , node1 , ... );
```

`scope` refers to a scope in the hierarchy, and `node` refers to a node specifier. If `scope` is omitted, the default is the current scope. If `node` is omitted, the default is all inputs, outputs, and inouts of the specified scope.

The word `node`, as used here, refers to nodes in a hierarchical structure; it has nothing to do with the word as it is used in analog simulation. A `node` can be one of the following:

Node	Signals That Enter the Database
"A"	All nodes (including inputs, outputs, and inouts) of the specified scope

"S"	Inputs, outputs, and inouts of the specified scope, and in all instantiations below it, except inside library cells.
"C"	Inputs, outputs, and inouts of the specified scope, and in all instantiations below it, including inside library cells
"AS"	All nodes (including inputs, outputs, and inouts) of the specified scope, and in all instantiations below it, except inside library cells)
"AC"	All nodes (including inputs, outputs, and inouts) in the specified scope and in all instantiations below it, even inside library cells)

For example, the following system task specifies all nodes in the current scope:

```
$shm_probe("A");
```

The following system task specifies all inputs, outputs, and inouts in the `alu` and `adder` modules:

```
$shm_probe(alu, adder);
```

The following system task specifies all inputs, outputs, and inouts in the current scope and below, excluding those in library cells; as well as all the nodes in the `top.alu` module and below, including those in library cells.

```
$shm_probe("S", top.alu, "AC");
```

Opening the SimVision Waveform Window

When working with waveforms, there are two use models:

- Using a Waveform window to view a post-simulation database after the simulation ends.
- Using a Waveform window for interactive waveform display so that you can view the waveforms as the simulation progresses.

Using the Waveform Window to View a Post-Simulation Database

When you use SimVision to view a post-simulation database, you are using it in PPE mode. For more information, see the *SimVision User Guide*.

To view the waveforms using SimVision after the simulator creates the database, do the following:

1. Open the viewer by typing
`simvision -waves`
2. Choose *File - Open Database*.
3. In the Open Database browser, navigate to the `.trn` file for your data.
4. Click *Open*.
5. In the Waveform window, choose *Windows - New Design Browser*.
6. In the Design Browser, select the signals you want to display.
7. Click *Send selected object(s) to target waveform window* to add the signals to the Waveform window.

Using the Waveform Window for Interactive Waveform Display during Simulation

To view the waveforms as the simulation progresses, do the following:

1. [Preselect the signals you want to view.](#)
2. [Run the simulation.](#)

If you do not preselect objects, you can use the Design Browser to probe signals and add them to the Waveform window.

Related Topics

- [Managing Databases](#)
- [Setting and Deleting Probes](#)

Displaying Debug Settings

While debugging, you can open databases, set probes, set breakpoints, set aliases, and so on. To display your current debug settings, do the following:

- If you are using the Tcl command-line interface, Use the appropriate modifier to display information. For most commands, this is the `-show` modifier. For example:

```
xcelium> database -show
```



```
xcelium> probe -show
```

Use the `alias` command without a modifier to display information about aliases you have set, as shown in the following example:

```
xcelium> alias
e      exit
f2     finish 2
h      history
xcelium>
```

- If you are using SimVision, use the commands on the Show menu. For example, to view information on breakpoints you have set, choose [Simulation - Show - Breakpoints](#).

Setting Variables

You can set Tcl variables to help debug your design. In addition to user-defined variables, the simulator includes several predefined Tcl variables that you can use to control various simulator features. You can do any of the following:

- Set a variable or change the value of a variable using the built-in Tcl `set` command:

```
xcelium> set abc 10 xcelium> set vlog_format %b
```
- Delete a variable using the `unset` command:

```
xcelium> unset abc
```
- Display a list of predefined simulation variables and their current values using the `help -variables` command:

```
xcelium> help -variables
```
- Display a list of all currently set variables using the `info vars` command:

```
xcelium> info vars
```

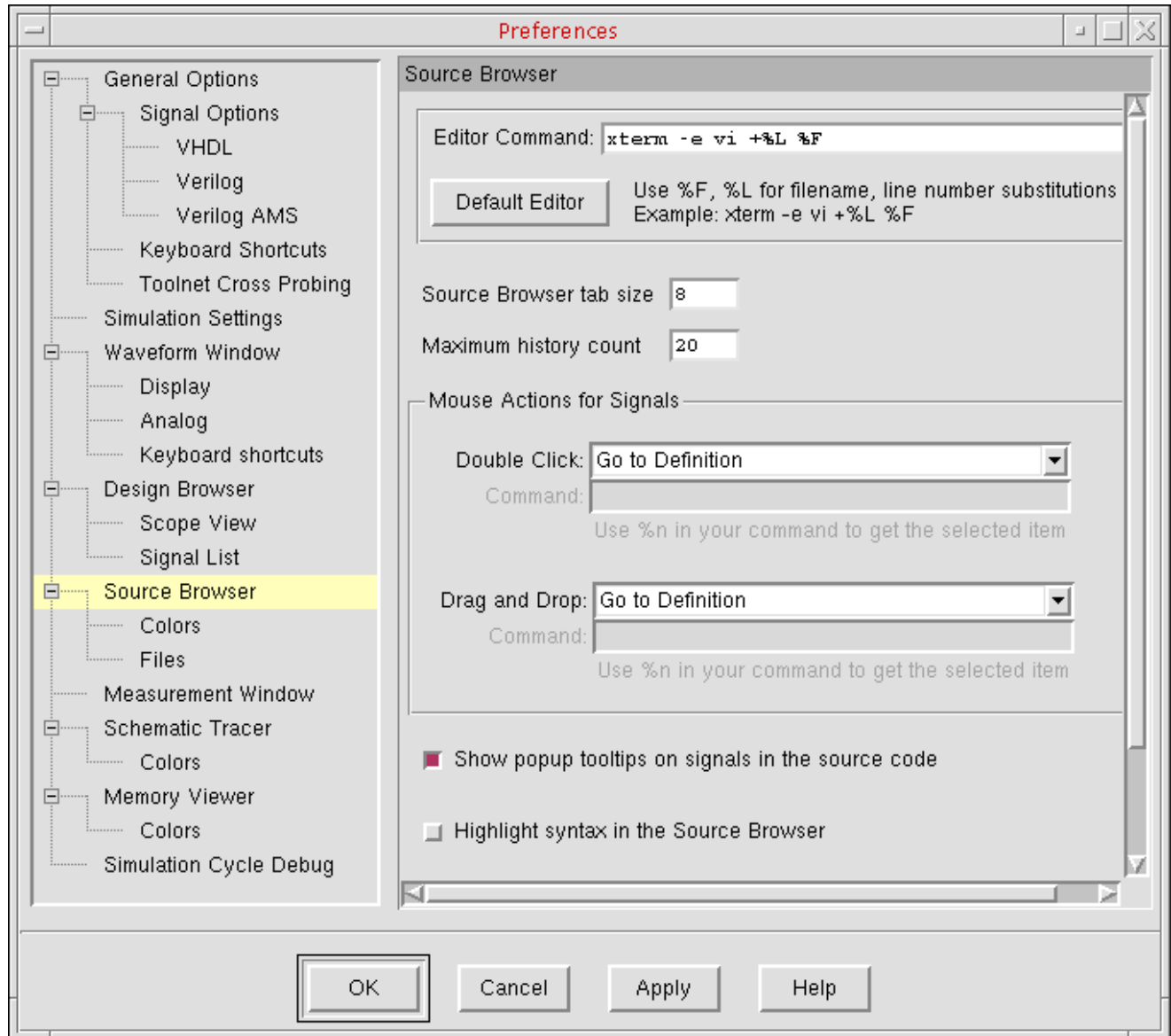
This command does not display variable values.

You can put variable definitions in an input file and then execute the commands in this file using the `-input` option when you start the simulator. You can also execute these commands using the Tcl [source](#) command or choosing *File - Source Command Script* after invoking the simulator.

Editing a Source File Using Your Own Editor

To specify the command to start the source file editor, do the following:

1. In any SimVision window, choose *Edit - Preferences*.
2. On the Preferences form, select **Source Browser** in the list on the left side of the window.



3. In the Editor Command field, type the command to start the text editor.
4. Click *OK*.

To start the editor to edit the current file in the Source Browser, do the following:

In the Source Browser window, click the Edit Source button on the toolbar:



The source file appears in your editor.

Searching the Source Code

You can use the SimVision Search Browser to find a text string or line in the source code.

The following sections explain the procedure.

- [Searching for a Line Number in the Source Code](#)
- [Searching for a Text String in the Source Code](#)

Searching for a Line Number in the Source Code

To find a particular line number in the source code, do the following:

1. In the SimVision Source Browser window, choose *Edit - Go to Line*.
2. In the Line field on the form that appears, type the line number you want.
3. Click *OK*.

An arrow appears and points to the specified line in the code.

Searching for a Text String in the Source Code

To search for a text string in the source code, do the following:

1. In the SimVision Source Browser window, choose [Edit - Text Search](#).
2. Use the Text Search form to specify the search string and parameters.
3. Click *Find Next*.

Saving and Restoring Your Simulation Environment

You can save and restore the current state of the debug environment.

- If you are using the Tcl command-line interface:

- Use the `save -environment` command to save the current state of the environment to a file:

```
xcelium> save -environment [ filename ]
```

This command generates a script containing Tcl commands to recreate breakpoints, databases, probes, and the values of Tcl variables. If `filename` is not specified, the script is written to standard output.

- To restore the environment, execute the script with the Tcl `source` command or use the `-input` option when you start the simulator.
- If you are using SimVision:
 - Choose *File - Save Command Script* to save the command script.
 - Choose *File - Source Command Script* to restore your debug settings.

When you source a script containing Tcl commands to restore a saved debug environment or choose *File - Source Command Script*, the software merges the debug settings in the script with your current debug settings.

Creating or Deleting an Alias

You cannot create an alias with the same name as a predefined Tcl command

To create an alias that you can use as shorthand for a command or series of commands,

- If you are using the Tcl command-line interface, use the `alias` command.
- If you are using SimVision, choose *Simulation - Create Command Alias*. To display current aliases, choose *Simulation - Show - Aliases*.

Getting a History of Commands

The `history` command lets you re-execute commands without retyping them. You also can use the `history` command to modify old commands--for example, to fix typographical errors.

To get a history of all the commands you typed, type `history`.

Managing Custom Buttons

If you are using SimVision, you can create custom buttons for commands and add them to the toolbar. You can assign one or multiple Tcl commands to a custom button. Customized buttons can automate your debugging tasks by letting you execute a series of Tcl commands with one click.

You can

- Create buttons that perform a specified function
- Edit the buttons to change the functions they perform
- Reorder the buttons on the toolbar
- Export the buttons to save them in a file
- Import a file of user-defined buttons

Xcelium Simulator Mixed-Signal Capabilities

Xcelium simulator with mixed-signal option can also be used to run AMS Designer features. It supports the simulation of languages such as Verilog-AMS and VHDL-AMS, SystemVerilog Real Number Modeling (SVRNM), SV and mixed-signal (MS) capabilities along with SPICE or Verilog-A, and other digital-centric mixed-signal technologies like low power and MS, code coverage with MS, and so on.

In addition to the features that are common to both Spectre AMS Designer Simulator and Xcelium simulator with the mixed-signal option, the following are some of the advanced capabilities of the Xcelium simulator with the mixed-signal option:

- [SystemVerilog Real Number Modeling](#)
- [SystemVerilog and AMS extensions](#)
- [Low Power and Mixed-Signal Verification](#)
- [SystemC Integration for Mixed-Signal Verification](#)
- [Incremental Elaboration for Mixed-Signal](#)

SystemVerilog Real Number Modeling

Real-Number Modeling (RNM) using the SystemVerilog language in a mixed approach borrowing concepts from the digital and analog domains that enables high-performance digital-centric, mixed-signal verification. You can use SV-RNM for the creation of accurate, high-speed Digital Mixed-Signal (DMS) models.

SystemVerilog Real Number Modeling uses real number ports defined in IEEE 1800-2012 standard supporting nettypes (built-in, UDT, UDR, and UDNs) and interconnects. Also, Cadence® provides a `EE_pkg` package that defines nettype, `EEnet` to describe analog impedance-based interactions between blocks in a SystemVerilog DMS environment. You can then examine advanced RNM features, SV port connections, and Connect Modules (CM) for AMS interactions.

Related Topics

- [Using Real Number Modeling in SystemVerilog](#)
- [SystemVerilog Interconnects](#)
- [SystemVerilog User Defined Nettype and Electrical Connections](#)
- [Enabling Net Aliasing for SV-UDN Connections](#)

Using Real Number Modeling in SystemVerilog

You can enable real number modeling in SystemVerilog designs using one of the following:

- Built-in nettypes with real data type and built-in resolution functions, equivalent to the `wreal` resolution functions
- User-defined nettype (UDN) and resolution function

To declare a nettype use the keyword `nettype`, and include a data type and optionally a resolution function. It can carry one or more values over a single net. A real value can be used to communicate voltage, current and other values between design blocks. You can also define a user-defined type (UDT), user-defined nettype (UDN), and user-defined resolution function (UDR).

Syntax:

```
nettype UDT UDN [ with UDR ];
```

Where,

- UDT is the datatype
- UDN is the nettype identified
- UDR is the resolution function

UDNs are SystemVerilog structures with a defined resolution function. Nets are used for structural connections and allow for the resolution of multiple drivers. Nets have a single, resolved value based on one or more drivers. Nettypes can only be driven through a continuous assignment, where each assignment represents a driver on the net. A nettype construct can carry one or more values over a single net. A real value can be used to communicate voltage, current, and other quantities between design blocks.

Resolution functions are used to determine the final value of a nettype. The return value of a resolution function is the nettype's datatype. It is called whenever there is a change on one or more drivers to a net. Nettypes that are defined to only have a single driver do not need to be defined with a resolution function. However, a nettype with a resolution function that has only one driver can still call that resolution function when a driver changes. Also, you can use a resolution function with a single-driver nettype. The resolution function can be used to define monitoring functionality and debug operations on the net.

Related Topics

- [Specifying Built-In SystemVerilog Nettypes](#)
- [Specifying User-Defined Nettype and Resolution Function](#)

Specifying Built-In SystemVerilog Nettypes

Cadence provides a pre-compiled SystemVerilog (SV) Package (`cds_rnm_pkg`) of wreal-like nettypes, available for import. This allows you to reuse and migrate the Verilog-AMS code to SV.

You can enable real number modeling in SystemVerilog by creating a set of built-in nettypes with `real` (`scalar` or `typedef`) data type and built-in resolution functions, equivalent to the wreal resolution functions.

A built-in real SV nettype can be declared as follows:

```
nettype real nettype_identifier with builtin_res_func;
```


Where:

`nettype_identifier` is the identifier that you use for nettype and `builtin_res_func` is a Cadence built-in resolution function. For example:

```
nettype real wrealavg with CDS_res_wrealavg
```

You can also use another name for an existing built-in nettype, as shown below.

```
nettype real wrealavg with CDS_res_wrealavg; //declare a built-in nettype "wrealavg"
nettype wrealavg myWrealAvg; //rename wrealavg to myWrealAvg
```

The SV built-in resolution functions and their equivalent wreal types are listed in the following table.

Built-In Resolution Functions

Built-in Resolution Function	Equivalent Wreal Type
CDS_res_wrealldriver	wrealldriver
CDS_res_wreal4state	wreal4state
CDS_res_wrealmin	wrealmin
CDS_res_wrealmax	wrealmax
CDS_res_wrealsum	wrealsum
CDS_res_wrealavg	wrealavg

Once a built-in nettype has been declared, a net of the built-in nettype can be used just like a net of any other user-defined nettype (UDN) as governed by the SV LRM.

```
nettype real wrealavg with CDS_res_wrealavg; //declare a built-in nettype "wrealavg"
wrealavg x; //declare a singular net of nettype "wrealavg"
wrealavg x [0:3]; // declare a 4-element array of nets of nettype "wrealavg"
```

You can easily port existing wreal models to SV in one of the following ways:

- By importing the `cds_rnm_pkg` package that defines a set of built-in nettypes that are equivalent to the typed wreal nets. See [Example 2](#).
- Using the `xrun` command-line option, `-wreal`. When `wreal` is used as a datatype in the design, you can use this option without using `cds_rnm_pkg`. The tool then internally imports the `cds_rnm_pkg` definition for `wreal`. See [Example 1](#).

Example 1

In the following example, in the design `wreal` type is used and in the `xrun` command-line the value, `wrealavg` is passed to this type using the `-wreal` option.

```
xrun -clean top.sv -wreal wrealavg -rnm_info -rnm_tech -dms_concat

module real_model(x);
input x [0:3];
wreal x [0:3];
child C(x);
endmodule
```

Example 2

The following simple Verilog-AMS `wreal` module

```
module real_model(x);

    input x [0:3];
    wrealavg x [0:3];
    child C(x);
endmodule
```

can be ported to use SV nettypes by simply importing the `cds_rnm_pkg` package, as shown below.

```
import cds_rnm_pkg::*;

module real_model(x);

    input x [0:3];
    wrealavg x [0:3];
    child C(x);
endmodule
```

The package is included in the Xcelium installation

at `$INSTALL_ROOT/tools/affirma_ams/etc/dms/cds_rnm_pkg.sv`.

When the software is installed, the package is compiled into a new `INSTALL_ROOT/tools/affirma_ams/etc/dms/dmsLib` directory. The package is automatically appended to the `xrun` command line and is available for import in all SV modules that `xrun` is compiling.

You can use the new `EE_pkg` package to port existing `wreal` models to SV. The package is included in the installation at `$INSTALL_ROOT/tools/affirma_ams/etc/dms/EE_pkg.sv`. When the software is installed, the package is compiled into the `$INSTALL_ROOT/tools/affirma_ams/etc/dms/dmsLib` directory.

The new `EE_pkg` package contains the definition of `EE` nettype and resolution. It provides the user-

defined type `EEnet` that handles pins that are structures (`EEstruct`) of three real values {V,I,R} (Voltage, Current, Resistance). `EEnet` models a combination of a series of voltage sources and resistors and a parallel current source and defines the resolution function to generate a resolved voltage from any useful combination of the inputs.

You can also refer to the examples related to electrical equivalent (EE) controlled elements for use with `EE_pkg` to model the electrical operations in SV in

the `$INSTALL_ROOT/tools/affirma_ams/etc/dms/dmsLib/EE_pkg_examples` directory. See the `Readme.txt` file in the `EE_pkg_examples` directory for more information on the examples.

Related Topic

- [Specifying User-Defined Nettype and Resolution Function](#)

Specifying User-Defined Nettype and Resolution Function

Like the built-in real SV nettype, a user-defined nettype (UDN) is declared with the keyword `nettype`, and includes a data type and optionally a resolution function. For example, the following declares a user-defined nettype `wTsum` with data type `T`:

```
nettype T wTsum;
```

Here, `T` is a `struct` data type with real fields. For example:

```
typedef struct {  
    real field1;  
    real field2;  
  
} T;
```

The valid data types for the nettypes are scalar reals and unpacked structures containing real, logic, bit, integer, and enum data types. Static arrays of unpacked structure nettype with or without resolution functions are supported. Other data types are not supported.

To use `longint` and `shortreal` datatypes in structured UDNs, you must use the `extended_struct_UDN` option.

Use the `with` keyword to optionally specify a resolution function to resolve the driven value of nets of user-defined nettype, as shown below.

```
nettype T wTsum with Tsum;
```

The resolution function of a nettype with a data type `T` is an SV function with a return type of `T`. The

function has a single input argument whose type is a dynamic array of elements of type `T`. A resolution function neither resizes the dynamic array input argument nor writes to any part of it.

Calls to only the following system tasks or functions are supported inside the resolution function:

<code>\$bitstoreal</code>	<code>\$realtobits</code>	<code>\$signed</code>
<code>\$cast</code>	<code>\$rttoi</code>	<code>\$itor</code>
<code>\$time</code>	<code>\$realtime</code>	<code>\$display</code>
<code>\$fdisplay</code>	<code>\$fwrite</code>	<code>\$bits</code>
<code>\$size</code>	<code>\$left</code>	<code>\$right</code>
<code>\$high</code>	<code>\$increment</code>	<code>\$fatal</code>
<code>\$warning</code>	<code>\$info</code>	<code>\$finish</code>
<code>\$error</code>	<code>\$dimension</code>	<code>\$sformatf</code>
<code>\$unsigned</code>	<code>\$stime</code>	<code>\$write</code>
<code>\$typename</code>	<code>\$low</code>	<code>\$resolved_wudn_name</code>
<code>\$resolved_wudn_drivers</code>	<code>\$test\$plusargs</code>	<code>\$value\$plusargs</code>

You can use the `-rnm_relax` option of `xrun` to access the global variables inside the resolution functions.

The `$resolved_wudn_name()` system task returns a string containing the hierarchical name of the simulated user-defined nettype(UDN) being simulated. It always returns the highest level of the net-segment in the following situations: explicitly typed UDN, interconnect (explicit) type, and wire (implicit interconnect) type. It can take a single string argument that is a hierarchical scope. If no scope is given it returns the name of the top-most net that is part of the collapsed net by traversing upwards. If a scope is given it will no traverse past that scope and will return a hierarchical net name that is at that scope or below.

Unpacked arrays of nets of user-defined nettype (both with and without user-defined resolution functions) are supported. For unpacked arrays of nets of user-defined nettype, each element of the array is considered as an atomic net.

Array sliced and concatenated connections to user-defined nettype with user-defined resolution function and nettype module ports are supported (see examples below).

`$resolved_wudn_drivers()` takes the driver array and an index and returns a string for the driver name.

Declaring a packed, multi-dimensional, or dynamic array of nets of user-defined nettype results in an error.

The resolution function of any net of a user-defined nettype is activated at time 0 at least once. This activation occurs even for nets with no drivers or when there is no value change on the drivers at time 0.

If a net has been defined as an unpacked array of nets (and `nettype` is defined at the element level), the resolution function is invoked for each element of the array. This means that if the array has five elements, the resolution function will be invoked five times at 0 simulation time.

The initial value of a net with a user-defined nettype is set before an `initial` or `always` procedure is started and before the activation of the time 0 resolution call. If no initializer is applied, the default initialization value for a net with a user-defined nettype is the default value defined by the data type. For a net with a user-defined nettype of `struct` data type, any initialization expressions for the members within `struct` are applied.

Assignment to nets of user-defined nettype is done using continuous assignments. Continuous assignments can be made with a declaration assignment or a continuous assignment statement.

A continuous assignment to an atomic net cannot drive a part of the net; it drives the entire net as per its nettype. Therefore, the left-hand side of a continuous assignment to a net of a user-defined nettype does not contain any indexing or select operations to the data type of `nettype`. In case of arrays of nets of user-defined nettype, indexing is done on the complete array because `nettype` is defined at the element level and not at the array level. However, no value can be written to the member of the element of the structure through the assignment statements targeting that member explicitly. For example:

```
assign dr_1[0] = T'{0.5, 0.5}; // legal assignment
assign dr_1 = 'T'{0.5, 0.5}, T'{2.3,3.4}}; // legal assignment
assign dr_1[0].field1 = 0.5 ; // illegal assignment
```

Assignment delays and net declaration delays are supported on the nets of user-defined nettypes and static arrays of unpacked structure nettype.

If the internal and external connections of a port are nets of user-defined nettypes and they are of matching nettype, they are merged into a single simulated net. Otherwise, an error is generated. If only one of the two connections is a net of user-defined nettype, then the connections must have matching data types; the port must be input or output, and the connection is treated as a continuous assignment from source to sink.

The following Tcl commands are supported on the user-defined nets and static arrays of unpacked structure nettype:

- `describe`

- `value`
- `drivers`
- `deposit`
- `probe -screen`
- `probe -shm`

SV-RNM (built-in SV nettypes and user-defined nettypes) is supported for the following in the AMS-CPF flow:

- Built-in SV-wreal nettypes defined in `cds_rnm_package`. For example `wrealldriver`, `wreal4state`, `wrealmin`, `wrealmax`, `wrealsum`, and `wrealavg`.
- SV-RNM interconnect nets that coerce to SV-wreal.
- User-defined nettypes with built-in resolution functions. Following built-in resolution functions are supported:
 - `CDS_res_wrealldriver`
 - `CDS_res_wreal4state`
 - `CDS_res_wrealmin`
 - `CDS_res_wrealmax`
 - `CDS_res_wrealsum`
 - `CDS_res_wrealavg`
- Array of built-in wreal nettypes.

SV-RNM is supported for the following CPF features:

- Creating power domain on blocks with real-valued signal
- Power Smart IE selection
- Interaction of power intent device and connect module
- Power intent propagation on mixed-signal boundary
- Dynamic compatibility check on mixed-signal boundary
- Power corruption on a real-valued signal
- IE power supply access from real-valued side

- CPF-controlled SV-Wreal/Interconnect power supply
- Real-valued signal driving a power event

Examples

Following is an example of arrays of net of structure nettypes without the resolution function:

```
// user-defined data type T
typedef struct {
    real field1;
    real field2;
}T;

// A nettype declaration with data type and resolution function
nettype T wTsum;

module top;
wTsum w[2];
T myvar[2];

assign myvar = w;
driver1 d1(w);
receiver1 r1(w);
endmodule

module receiver1(rec_1);

output rec_1[2];
wTsum rec_1[2];
initial
    #1 $display("sum = %p flag = %p \n", rec_1[0], rec_1[1]);

endmodule

module driver1 (dr_1);
output dr_1[2];
wTsum dr_1[2];
assign dr_1[0] = T'{0.5, 0.5};
assign dr_1[1] = T'{1.5, 1.5};

endmodule
```

Following is an example of arrays of net of structure nettypes with the resolution function:

```
// user-defined data type T

typedef struct {
    real field1;
    real field2;
}T;

// user-defined resolution function Tsum
function automatic T Tsum (input T driver[]);
begin

    Tsum.field1 = 0.0;
    foreach (driver[i])
    begin

        $display("driver[%d]{%f, %f}",i, driver[i].field1, driver[i].field2);
        Tsum.field1 += driver[i].field1;
        Tsum.field2 += driver[i].field2;
    end

    $display("Tsum{%f, %f}", Tsum.field1, Tsum.field2);

end

endfunction

// A nettype declaration with data type and resolution function
nettype T wTsum with Tsum;

module top;
wTsum w[2];
T myvar[2];

assign myvar = w;
driver1 d1(w);
driver2 d2(w);
receiver1 r1(w);
endmodule

module receiver1(rec_1);

output rec_1[2];
    wTsum rec_1[2];
    initial

        #1 $display("sum = %p flag = %p \n", rec_1[0], rec_1[1]);

endmodule
```



```
module driver1 (dr_1);
output dr_1[2];

    wTsum dr_1[2];
    assign dr_1[0] = T'{0.5, 0.5};
    assign dr_1[1] = T'{1.5, 1.5};
endmodule

module driver2 (dr_2);

input dr_2[2];

    wTsum dr_2[2];
    assign dr_2[0] = T'{2.5, 2.0};
    assign dr_2[1] = T'{3.5, 3.0};

endmodule
```

Following is an example of a static array of unpacked structure nettype without resolution function:

```
module top;

typedef enum {ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX} enum_t;
typedef struct {
    real r;
    integer i;
    bit b;
    enum_t e;
} instruction;

nettype instruction net_r_inst;

instruction I1[0:3] = '{instruction'{1.0, 1, 1, ONE},instruction'{2.0, 2, 1, TWO},
instruction'{3.0, 3, 1, THREE}, instruction'{4.0, 4, 1, FOUR}};

instruction I2[0:3] = '{instruction'{11.0, 11, 1, ONE},instruction'{12.0, 12, 1, TWO},
instruction'{13.0, 13, 1, THREE}, instruction'{14.0, 14, 1, FOUR}};

net_r_inst I3[0:3] = '{instruction'{21.0, 21, 1, ONE},instruction'{22.0, 22, 1, TWO},
instruction'{23.0, 23, 1, THREE}, instruction'{24.0, 24, 1, FOUR}};

always @(I3[1].r) begin
$display("\n(%f NS) => I3 with %p = {p}", $time, I3);
$display("(%f NS) => I3[0].r = %f", $time(), I3[0].r);
$display("(%f NS) => I3[1].r = %f", $time(), I3[1].r);
$display("(%f NS) => I3[2].r = %f", $time(), I3[2].r);
$display("(%f NS) => I3[3].r = %f", $time(), I3[3].r);
end
```

```
endmodule
```

Following is an example of a static array of unpacked structure nettype with resolution function:

```
module top;

typedef enum {ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX} enum_t;
typedef struct {

    real r;
    integer i;
    bit b;
    enum_t e;
} instruction;

function automatic instruction FUNC(input instruction drivers[]);

begin

FUNC = '{1.1, 1, 0, ZERO} ;
    foreach (drivers[j]) begin
        FUNC.r += drivers[j].r;
        FUNC.i += drivers[j].i;
        FUNC.b = drivers[j].b ;
        FUNC.e = drivers[j].e ;
    end

end

endfunction

nettype instruction net_r_inst with FUNC;

instruction I1[0:3] = '{instruction'{1.0, 1, 1, ONE},instruction'{2.0, 2, 1, TWO},
instruction'{3.0, 3, 1, THREE}, instruction'{4.0, 4, 1, FOUR}};

instruction I2[0:3] = '{instruction'{11.0, 11, 1, ONE},instruction'{12.0, 12, 1, TWO},
instruction'{13.0, 13, 1, THREE}, instruction'{14.0, 14, 1, FOUR}};

net_r_inst I3[0:3] = '{instruction'{21.0, 21, 1, ONE},instruction'{22.0, 22, 1, TWO},
instruction'{23.0, 23, 1, THREE}, instruction'{24.0, 24, 1, FOUR}};

assign I3 = I1 ;
assign I3 = I2 ;

always @(I3[1].r) begin
    $display("\n(%f NS) => I3 with %%p = {%p}", $time, I3);
    $display("(%f NS) => I3[0].r = %f", $time(), I3[0].r);
    $display("(%f NS) => I3[1].r = %f", $time(), I3[1].r);
    $display("(%f NS) => I3[2].r = %f", $time(), I3[2].r);
```

```
    $display("(%f NS) => I3[3].r = %f", $time(), I3[3].r);
end
endmodule
```

Following is an example of SystemVerilog array slices for user-defined nettype port connection:

```
//Declaration of nettype
package multiple_drive_wire;

typedef struct{
    real x;
    real y;
} user_wire;

function automatic user_wire resolve(user_wire driver[]);
    $display("Resolve");

    foreach(driver[i])begin
        resolve.y += driver[i].x;
    end
endfunction // resolve

nettype user_wire user_wire_multi_drv with resolve;

endpackage // tb

//Declaration of module with nettype port
module foo(inout user_wire_multi_drv a[1:0]);

    real x1,x2;

    assign a[0] = '{x1,0};
    assign a[1] = '{x2,0};

    initial begin
        x1 = 0.00;
        x2 = 0.00;

        for(int ii=0; ii < 10; ii++)begin

            x1 = 2.00*ii;
            x2 = 2.00*ii+1.00;
            #100;
        end

    end

endmodule // foo
```

Modules with nettype ports using array slices can be connected using the following methods:

```
import multiple_drive_wire::*;

module top;

    //interconnect bar[3:0]; --Standard declaration

    //wire bar[3:0]; --Cadence RNM tech declaration (use -rnm_tech option to compile)

    //user_wire_multi_drv bar[3:0]; --Explicit declaration

    foo u0(.a(bar[1:0]));          //array sliced port connect

    foo u1(.a({bar[1],bar[0]})); //Concatenated Port Connect

endmodule // tb
```

Related Topics

- [System Verilog Real Number Modeling Overview \(Video\)](#)
- [Debug Support for Resolution Functions \(Video\)](#)

Resolving Wreal Nets of Built-In Nettype

The resolution function of wreal nets of built-in nettype runs at time 0 regardless of the state of its drivers to bring them into compliance with the SystemVerilog nets of nettype, as described in the IEEE 1800-2012 standard.

Consider the following example:

```
module top();
    wrealldriver x;

    real a;
    real b;

    initial
        begin
            a = 0;
            b = 0;
            #1 a = 1.1;
            #1 b = `wrealZState;
            #1 $finish();
        end
end
```

```
always
begin
    $display("%t: x = %f", $time(), x);
    @(x);
end

assign x = a;
assign x = b;
endmodule
```

In the above example, `x` is a net which is declared as nettype `wreal1driver` and has two drivers `a` and `b` both driving 0.0 at time 0.

Running the above example results in the following output:

```
0: x = 0.000000
0: x = `wrealXState
xmsim: *W,WRMNZD: top.x: wreal net has multiple non-Z drivers (Time: 1 NS + 1).
2: x = 1.100000
Simulation complete via $finish(1) at time 3 NS + 0
```

In the above output, `x` resolves to `wrealXState` at time 0.

Handling Port Connections of Nettypes

Port Connections of nets of built-in and user-defined nettypes are governed by the port connection rules in the SystemVerilog (SV) LRM (section 23.3.3).

Both the upper and lower connections are nets of built-in nettype

- If both are singular nets of built-in nettype then they must have a matching nettype as governed by the matching nettype rules in section 6.22.6 of the LRM. The port shall be merged into a single simulated net.
- If either of the net is an array of nets of built-in nettype, their elements must have matching nettype as described in section 6.22.6 of the LRM. The port shall be merged into a single simulated net.

You can use the `xmelab/xrun -nettype_port_relax` command-line option to allow for relaxed port compatibility rules for connections of built-in nettypes. If this option is specified the following rules will govern connections of built-in nettype to built-in nettype:

- If they are both singular nets then their resolution functions must be the same. The port shall be merged into a single simulated net.
- If either is an array they must have an assignment-compatible data type as described in section 6.22.3 of the LRM. Size mismatches are allowed when connecting to other nettypes or wreals. In addition, the resolution functions of their element nettypes must be the same. The port shall be merged into a single simulated net.

One is a net of built-in nettype and the other is a wreal net

- If both are singular nets then the resolution function of the net of built-in nettype must match the wreal type from the table shown above. The port shall be merged into a single simulated net.
- If either of the net is an array, the nets must have an assignment compatible data type as described in section 6.22.3 of the LRM. In addition, the resolution function of the element nettype of the net of built-in nettype must match the wreal type from [Table, Built-In Resolution Functions](#). The port shall be merged into a single simulated net.

These rules for wreal connections assume that wreal coercion has already taken place and that all generic wreals have been resolved to a type.

One is a net of built-in nettype and the other is a variable or expression

- If the net of built-in nettype is a singular net then the data type of the variable or expression must be a singular real. The port shall be of mode input or output and the connection shall be treated as a continuous assignment from source to sink.
- If the net of built-in nettype is an array then the upper and lower connections must have an assignment-compatible data type and the element data type of the non-net must be real. The port shall be of mode input or output and the connection shall be treated as a continuous assignment from source to sink.

Related Topic

- [Specifying User-Defined Nettype and Resolution Function](#)

Wreal Interaction with Built-In Real Nettypes and Real Nettypes with Resolution Functions

The following are some of the supported use cases of interactions between wreal and built-in real ntypes and real ntypes with resolution functions:

- Wreal with no resolution function is connected to a real nettype with no resolution function.

```
typedef real vtype;  
nettype vtype nt;  
wreal wr;
```

- A wreal with no resolution function is connected to a real nettype with a compatible built-in resolution function.

```
typedef real vtype;  
nettype vtype nt with CDS_res_wrealavg; //built-in resolution function  
                                CDS_res_wrealavgwreal wr;
```

- A wreal with built in resolution function is connected to a real nettype with a compatible built-in resolution function.

```
typedef real vtype;  
nettype vtype nt with CDS_res_wrealavg;  
wrealavg wr;
```

- A wreal with no resolution function is connected to a nettype with user-defined resolution function. In this case, the wreal is coerced to the nettype's user-defined resolution function.

```
typedef real T;  
function automatic T Tsum (input T driver[]);  
begin  
    Tsum=0.0  
    foreach (driver[i])  
        begin  
            Tsum +=driver[i];  
        end  
    $display( "Tsum {%f}" , Tsum);  
end  
endfunction
```

```
nettype real nt with Tsum  
wreal wr;
```

Currently, wreals connected to incompatible (non-real) nettypes are not supported.

Related Topic

- [Using Real Number Modeling in SystemVerilog](#)

Wreal Interaction with Nets of Built-In Nettype

The following are some of the extended functionalities you can use with wreal designs and how it works with built-in nettypes:

- **Wreal Coercion:** In general, nets of built-in nettype behave in a similar manner to the wreal nets. Wreal coercion takes place before port compatibility checking, therefore, once the type of the port is fully known, port compatibility described in [Handling Port Connections](#) is considered.

Connecting an implied interconnect wire to a net of built-in nettype coerces that implied interconnect to a wreal net. The type of the coerced wreal net is determined by identifying the wreal type that corresponds to the net of built-in nettypes resolution functions described in [Table, Built-In Resolution Functions](#).

Connecting a net of built-in nettype to a generic wreal net coerces the type of the generic wreal net in the same manner that it coerces an implied interconnect. The final type of the generic wreal is determined by identifying the wreal type that corresponds to the net of built-in nettypes resolution function described in [Table, Built-In Resolution Functions](#).

If an array of nets of built-in nettype is connected to a collapsible concatenation of nets, all nets in the concatenation that are coercible to wreal is coerced. The type of the coerced wreal nets is determined by identifying the wreal type that corresponds to the net of built-in nettypes resolution function described in [Table, Built-In Resolution Functions](#). If any of the nets are not coercible they are not coerced and an R2L connect modules is inserted.

If an array of nets of built-in nettype is connected to a non-collapsible concatenation expression no coercion occurs, even if some of the elements of the concatenation would otherwise be coercible.

- **Discipline Resolution:** Discrete or continuous disciplines are not propagated to the nets of built-in nettype and discipline resolution does not propagate disciplines through them in either detailed or default discipline resolution.

- **Connect Module Insertion:** Insertion of Verilog-AMS wreal connect modules on nets of built-in nettype is allowed only through the `ie` card mechanism. The following connections are supported with an inserted connect module:
 - **Net of built-in nettype to electrical net.** If an `ie` card exists that applies to this connection, a VAMS connect module with one wreal port of the appropriate direction, one electrical port of appropriate direction, and `vsup` as defined in the applied `ie` card are inserted.
 - **Net of built-in nettype to logic net or variable.** If an `ie` card exists that applies to this connection, a VAMS connect module with one wreal port of the appropriate direction, one digital logic wire port of appropriate direction, and `vsup` as defined in the applied `ie` card are inserted.

If a connect module is needed on a net of built-in nettype and there is no `ie` card that would apply to that connection, an error is generated and no connect module is inserted.

- **Wreal Concatenation:** Connections of wreal concatenations to arrays of nets of built-in nettype are allowed as long as the concatenation only contains wreal nets (coerced or declared). The port collapses to a single simulated net. If the port cannot collapse, an error is generated.

The following connections are not supported:

- Inherited connections on nets of built-in nettype
- Bit-select, part-select, and indexed part-select of concatenation expressions

SystemVerilog Interconnects

SystemVerilog (SV) interconnect nets are specified using the keyword `interconnect` and can be used only in `net_lvalue` port expressions. These are also called explicit interconnects. In addition, there are interconnect nets, also called implicit interconnects, that are declared as wires but through analysis the elaborator determines whether they should be treated as interconnect nets. Implicit interconnect nets are not declared using the `interconnect` keyword. SV supports both types of interconnect nets.

An implicit interconnect is considered an interconnect net, if it meets all of the below criteria:

- The net is declared as `wire`, `tri`, `wand`, `triand`, `wor`, or `trior`
- The net is used only in `net_lvalue` port expressions

- The net is either singular or a packed/unpacked array with a single dimension
- The net is used in concatenation expressions as actual or formal of a non-collapsible port association
- The net does not connect to a variable or expression with any of the following data types:
 - string
 - event
 - chandle
 - unpacked struct that is not supported in the `nettype` declaration

In general, explicit interconnects can only be connected to nets. However, they can be connected to non-nets under the following conditions:

- If an interconnect connects to both non-nets and nets, the nettype of the interconnect is the nettype of its net connections:
 - If all its connections (both net and non-net) do not have assignment-compatible datatype, an error is generated
 - If its net connections do not have an equivalent nettype, an error is generated
- If an interconnect connects only to non-nets:
 - If the datatypes of all the non-net connections are not assignment-compatible, an error is generated
 - If all the non-nets have real datatype, the nettype of the interconnect is the built-in real nettype with the resolution `CDS_res_wrealldriver`
 - If all the non-nets have assignment-compatible non-real datatype, the nettype of the interconnect is an implied unresolved nettype with the datatype of the non-nets
 - If the non-nets have non-equivalent datatypes that are assignment-compatible, one of the datatypes is chosen for the implied unresolved nettype

Explicit interconnects are supported in the AMS CPF flow. For explicit interconnects, the following is supported:

- Power domain information on the interconnect net
- Creation of boundary port information on the interconnect net
- Power-smart IE (R2L/L2R/RL_Bidir, R2E/E2R/ER_Bidir) connections to the interconnect net
- Power state propagation on mixed-signal boundary

- Power domain voltage check on mixed signal boundary

The following is not supported:

- Power corruption on interconnect net
- Automatic power supply on interconnect nets
- Interconnect nets in power event control condition

Explicit interconnects can be connected to any port expression that adheres to the following port connection rules:

All port compatibility rules described below must still be followed once the type of interconnect is determined.

Port Connection Rules

A singular interconnect can connect to any of the following:

- Singular net of user-defined nettype
- Scalar electrical net
- Singular wreal
- Singular built-in logic net

Connection of a singular interconnect to an array port, or an array port to a singular interconnect is governed by the following rules:

- If an interconnect array is connected to an SV array net of user-defined nettype, both upper and lower port expressions must have the same number of elements. You can use the `xrun` option `-nettype_port_relax` to make the upper and lower port expressions have different number of elements.
- If an interconnect array is connected to a wreal array, the upper and lower port expressions may have different numbers of elements.
- If an interconnect array is connected to an electrical bus, the upper and lower port expressions may have different numbers of elements.
- If an interconnect array is connected to a packed built-in logic net array or structure, the upper and lower port expressions may have different numbers of elements.
- If an interconnect array is connected to an unpacked built-in logic net array, both upper and

lower port expressions must have the same number of elements.

- An interconnect array cannot be connected to an unpacked built-in logic net structure.
- If an interconnect array is connected to a variable array, both upper and lower port expressions must have the same number of elements.
- If an interconnect array is connected to a non-collapsible array expression, both upper and lower port expressions must have the same number of elements.

Port Collapsing Rules

When a port connection contains an interconnect net, the following collapsing rules apply:

- If either the inner or outer net is an interconnect and the other is a net with a declared nettype (either user-defined or built-in), the dominating net is the net with a declared nettype.
- If both inner and outer nets are interconnects, then the outer net is the dominating net.

However, if you use the `-delay_wire_dominant` option, the port collapsing rules would be as follows:

- If either inner or outer net is an interconnect net, and the other is a net with a declared nettype, then:
 - If both have a specified delay value, the dominating net is the net with a declared nettype.
 - If only one of them has a delay value, the dominating net is the one with the delay value.
 - If neither of them has a delay value, the dominating net is the net with a declared nettype.
- If both the inner and outer nets are interconnect:
 - If both have a specified delay value, the dominating net is the outer net.
 - If only one of them has a delay value, the dominating net is the one with the delay value.
 - If neither of them has a delay value, the dominating net is the outer net.

Interconnect Connections to VHDL Ports

Explicit interconnects can be connected to VHDL real ports by coercing the interconnect to `wreal`. In addition, explicit interconnects can be connected to VHDL real ports by coercing the interconnect net to a built-in real nettype. The resolution for explicit interconnects is the same as implicit interconnects. That is, if an explicit interconnect is also connected to a `wreal` or a net of built-in nettype, it takes that type, otherwise, it gets the default type.

For nets that have a mix of `wreal` and VHDL real signals, the resolution may change depending on where the drivers of the mixed language net exist. If all the drivers are in VHDL, the resolution is also done in VHDL. If the net has any Verilog drivers, then the resolution is done in Verilog (`wreal` resolution). If the VHDL and `wreal` resolution functions do not match, the behavior may be different based on the design configuration.

Explicit interconnects connected to VHDL ports are coerced to electrical. In addition, an interconnect connected to a scalar or vector `STD_LOGIC` port is coerced to a packed array of logic equivalent to a Verilog logic wire.

Discipline Resolution

Explicit interconnects support default, detailed, and block-based discipline resolution.

Net Delays on Interconnects

The AMS Designer simulator supports a single value net delay on explicit interconnects. For implicit interconnects having multiple delay values, the following rules apply:

- If the interconnect is coerced to a non-logic wire type (like `real` or `struct`) the net delay is converted to a single net delay that is the largest of the delay values specified.
- If the interconnect is coerced to a logic wire type, the multiple delay values are preserved.

When ports are collapsed, only the delay on the dominating net is applied. Since the typed net is always the dominating net, net delays on interconnects may not be applied. For more information on how interconnect ports collapse, see [Port Collapsing Rules](#).

In addition, when an interconnect (explicit or implicit) is coerced to electrical, the net delay is ignored.

Related Topic

- [Handling Port Connections of Nettypes](#)

SystemVerilog User Defined Nettype and Electrical Connections

You can connect a design that has SystemVerilog user-defined nettype (UDN) to electrical net connections by using the built-in nettype or custom UDN to electrical bidirectional connect modules to set up a SystemVerilog Real Numbered Model UDN and a Verilog-AMS (or SPICE/Spectre) electrical port connected on a mixed-signal net design.

To set up a SystemVerilog UDN-Electrical connection, you can do the following:

- Use the built-in nettype (EEnet) declared in the `EE_pkg.sv` package file to connect electrical ports (Verilog-AMS or SPICE/Spectre). These connections are done by inserting built-in Connect Modules (`EEnet_2_E.svams`, `E_2_EEnet.svams`, `EEnet_to_E_bidir.svams`).
- Create custom user-defined nettype (UDN) with resolution function and custom Connect Modules to insert between custom UDN and electrical connections.

Related Topics

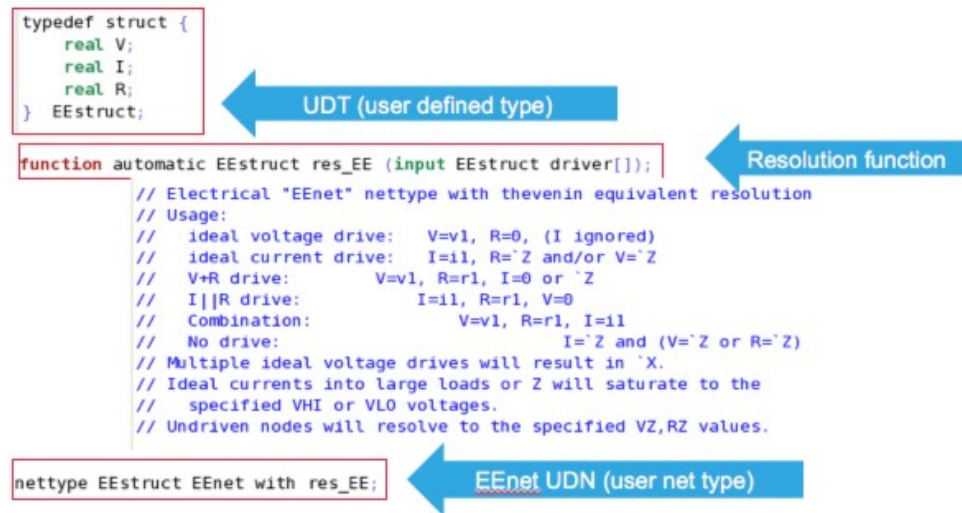
- [Using the Built-In EE Package Connect Modules for UDN-to-Electrical Connections](#)
- [Using Custom User-Defined Nettype and Connect Modules for UDN-to-Electrical Connections](#)

Using the Built-In EE Package Connect Modules for UDN-to-Electrical Connections

Cadence provides the `EE_pkg.sv` package file that contains the definition of `EEnettype` and resolution. It provides the user-defined type, `EEnet` that handles pins that are structures (`EEstruct`) of three real values {V,I,R} (Voltage, Current, Resistance). `EEnet` models a combination of a series of voltage sources and resistors and a parallel current source and defines the resolution function to generate a resolved voltage from any useful combination of the inputs.

The `ee_pkg.sv` file is available in the `your_install_dir/tools.lnx86/affirma_ams/etc/dms` directory. This file contains a user-defined type (UDT) "`EEstruct`", a user-defined resolution function (UDR) "`res_EE`", and a user-defined nettype (UDN) "`EEnet`".

The Built-in `ee_pkg.sv` File



To automatically insert the built-in connect module for a EENet-to-Electrical Connection, you must do the following:

- Import the `EE_pkg` file and declare the `EENet` type in your SV design. For example:

```
import EE_pkg::*;
module VsrcG(P,vval,imeas);
    inout EENet P;
    input real vval;
    output real imeas;
    ...
endmodule
```

The above example is provided in the `VsrcG.sv` file available in the `your_install_dir/tools.lnx86/affirma_ams/etc/dms/EE_pkg_examples` directory.

- Declare the electrical object in the `.vams` file. For example:

```
`timescale 1ns/1ps
`include "disciplines.vams"
module EVRsrc(P,N,vval,rval,imeas);
    inout P,N; electrical P,N;
    input wreal vval,rval;
    output wreal imeas;
    ...
endmodule
```

```
Endmodule
```

- Specify how you want the tool to map the UDN ports to the Electrical connection ports. The following example illustrates a top module instantiating `VsrcG` module having `EEnet` port connecting to Electrical port in `EVRsrc`.

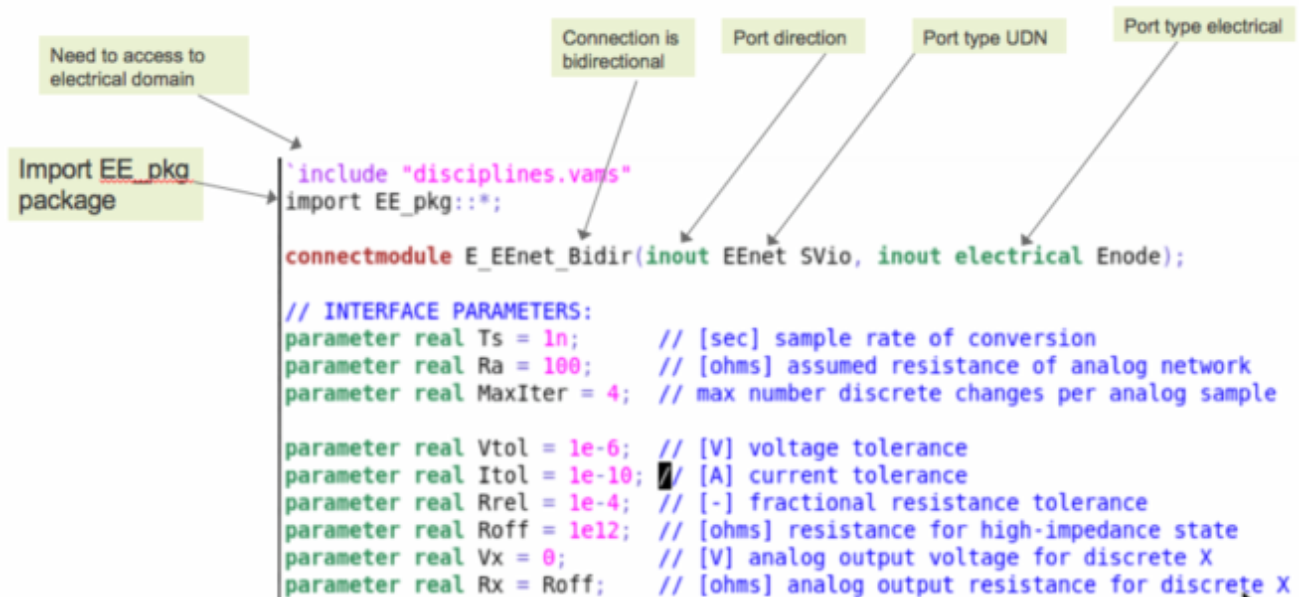
```
module top();
...
///
VsrcG vlda ( LDda, V1 , Ivlda); //Instance with EEnet Port
EVRsrc rlda (LDda, RAdA, 0.0 , R1 , Irlda) ; //Instance with Electrical port
...
endmodule
```

The tool automatically inserts the appropriate built-in connect module between the EEnet-electrical ports. The following are the in-built connect module specification files:

- **EEnet_to_E.svams**: This is an EEnet port to electrical connect module example. It connects an EEnet port to an electrical port in the SV domain.
- **E_to_EEnet.svams**: This is an electrical to EEnet port connect module example. It connects an Electrical port to an EEnet port in the SV domain.
- **EEnet_to_E_bidir.svams**: This is a bidirectional SystemVerilog-AMS connect module example. It can connect an Electrical port to an EEnet port in the SV domain.

These connect modules have appropriate port types and directions for use. For instance, the following example illustrates a bidirectional Electrical to EEnet port connect module:

Bidirectional Electrical to EEnet Connect Module



By default, the built-in nettype is specified in the AMSD control block. However, you can modify the values, if required. For more information, see [Specifying UDN in the AMSD Control Block](#).

Related Topic

- [SystemVerilog User Defined Nettype and Electrical Connections](#)

Using Custom User-Defined Nettype and Connect Modules for UDN-to-Electrical Connections

In addition to the built-in connect modules, you can create custom user-defined nettype (UDN) with resolution function and connect modules (user-defined connect modules) to connect a design that has SystemVerilog user-defined nettype (UDN) to electrical net connections.

To use custom UDNs, do the following:

1. [Create a Custom User-Defined Nettype](#): Create your own `*.sv` package file and define the custom user-defined nettype (UDN) with resolution function in this file.
2. (Optional) [Creating Custom Connect Module](#): Create custom connect module that contains connect module definitions and discipline definitions to suit your design requirements.

3. **Specify UDN in the AMSD Control Block:** Specify the user-defined nettype and connect module parameters for the UDN-Electrical connections in the AMSD control file.

Creating Custom User-Defined Nettype

You can create your own *.sv package file and define the custom user-defined nettype (UDN) with resolution function in this file.

A UDN is declared with the System Verilog keyword `nettype`, and must include a data type with a resolution function. Use the `with` keyword to specify the resolution function. For example, the following declares a user-defined nettype, `myNT` with datatype `myT` and a UDR, `RESFUNC`:

```
nettype myT myNT with RESFUNC;
```

The following is an example of a custom UDN with resolution function:

```
// udn_package
//

package udn_package;

    typedef struct {
    real r;
    } myT;

    function automatic myT RESFUNC(input myT drivers[]);
begin

    RESFUNC.r = 0.0;
    foreach (drivers[j]) begin

        // $display("\nRESFUNC : drivers[%0d].r = {%0f} ", j, drivers[j].r);

        if (drivers[j].r === `wrealZState) continue;

        else RESFUNC.r += drivers[j].r;

    end

end

endfunction

nettype myT myNT with RESFUNC;

endpackage
```

Creating Custom Connect Module

You can create custom connect module that contains connect module definitions and discipline definitions to suit your design requirements. And, you must save the file with the SystemVerilog-AMS file extension `.svams`.

The following must be specified in the custom connect module file:

- Import the custom `udn_package` (UDN). This is the `.sv` file in which you have defined your custom user-defined type and user-defined resolution function.
- Specify the custom connect module name and logic. The name of the custom connect module must be based on the port direction. For instance, the following is an example of an Electrical-to-UDN connection, and the name of the custom module is `E_2_myNT`.

Bidirectional SystemVerilog-AMS Connect Module

```
`include "disciplines.vams"

import udn_package::*;

connectmodule E_2_myNT (Ain, Dout); // input electrical Ain, output myNT Dout

    input Ain;
    electrical Ain;
    output Dout;
    myNT Dout;
parameter real vss=0 ;// from (-inf:inf); // Voltage of negative supply
...

endconnectmodule
```

top.sv

The following example (`top.sv`) illustrates how a custom connect module is inserted between the custom UDN `myNT` and the electrical port `sigout`, which is declared in `elect_sine_driver` (See [Example, analog.vams](#)).

```
import udn_package::*;

module top;
  myNT ntl;
  real myr;
  elect_sine_driver sine_driver1 (ntl); // 0 to 10v sine wave driver
  assign myr = ntl.r;
endmodule
```

analog.vams

```
`timescale 1ns / 1ns

`include "disciplines.vams"

module elect_sine_driver (sigout);

  output sigout;
  electrical sigout;
  electrical gnd;
  ground gnd;

  vsource #(.type("sine"), .ampl(5.0), .dc(5.0), .freq(1.0M)) v0 (sigout, gnd);

endmodule
```

Specifying UDN in the AMSD Control Block

After you declare the custom UDN and connect module, you must specify the user-defined nettype and connect module parameters for the UDN-Electrical connections in the AMSD control file. It is defined using the `ie` card statement in an `amsd` block.

The following example defines the connect module insertion for the custom nettype `myNT`. You can also customize parameters named `Ts`, `Ra`, `MaxIter`, `Vtol`, `Itol`, `Rrel`, `Roff`, `Vx`, `Rx`.

The AMSD Control File

```
//AMSD CONTROL FILE
```

```
amsd {  
  ie vsup=5.0 nettype="myNT"  
}
```

vsup must be specified for binding CM.

The following parameters of a connect module are customizable via ie card: vsup, Ts, Rs, Maxlter, Vtol, Itol, Rrel, Roff, Vx, and Rx.

Specify the user-defined nettype with the Electrical domain.

A sample AMSD control file, `amscf.scs` is available at, `< your_install_dir >/tools/affirma_ams/dms/EE_pkg_examples.`

If you are using the built-in UDN, you do not have to explicitly specify the nettype parameter in the `ie` card statement. The tool automatically identifies the built-in UDN.

Related Topics

- [SystemVerilog User Defined Nettype and Electrical Connections](#)
- [Modeling Electrical Behavior in System Verilog - RNM](#)

Enabling Net Aliasing for SV-UDN Connections

Use the system function `$real_net_alias()` to alias a local net to a hierarchical net through a string reference within connect modules and regular modules.

In SV-UDN real net array connections, both the `real_net_reference` and `hierarchical_reference_string` arguments must be of the same nettype. And, the `$real_net_alias` system function does not support SV-UDN nets with struct type; and, returns zero value in this case.

Related Topic

- [Aliasing Nets to Hierarchical Nets in L2R/R2L Interface Elements](#)

Wreal Connections with Concatenated Nettypes to Array of Instances

For connections between concatenated wires and array of instances, the tool enables coercion of parent wires in concatenation expression to respective mapped nettype, if the wires are coercible. However, for the such connections to function correctly, RNM processing must be enabled using the `-rnm_tech` option.

Example 1

In the following example, the wires `in1` and `in2` in the concatenation expression are coerced to nettype `EEnet`. The tool checks if each bit or slice of net in the concatenation expression is compatible with the port connection.

```
# Design:
module test();
    wire in1;
    wire in2;
    test1 test1( .out({in2,in1}) );
endmodule

module test1( output [1:0] out );
    test1_1 test1_1[1:0](out);
endmodule

module test1_1( output EEnet out );
endmodule

# Log info:
RNM_INFO: test.in1 is coerced to nettype 'EEnet' with resolution 'res_EE'
<< -- coercion
RNM_INFO: test.in2 is coerced to nettype 'EEnet' with resolution 'res_EE'
<< -- coercion
RNM_INFO: test.test1.out is coerced to nettype 'EEnet' with resolution
'res_EE'
RNM_INFO: test.test1.test1_1[1].out is declared to nettype 'EEnet' with
resolution 'res_EE'
RNM_INFO: test.test1.test1_1[0].out is declared to nettype 'EEnet' with
resolution 'res_EE'
RNM_INFO: test.test2_1.in is coerced to nettype 'EEnet' with resolution
'res_EE'
```

```
RNM_INFO: test.test2_1.tes3.in is declared to nettype 'EEnet' with resolution
'res_EE'
RNM_INFO: test.test2_2.in is coerced to nettype 'EEnet' with resolution
'res_EE'
RNM_INFO: test.test2_2.tes3.in is declared to nettype 'EEnet' with resolution
'res_EE'

RNM Report Summary:
    RNM declared total: 4
    RNM coerced total: 3
```

Example 2

Similarly, if the width of a concatenation expression of parent nets is equal to the width of an individual child port multiplied by the size of an instance array, the tool splits the concatenation expression and connects each bit/slice of the parent net to the port of one instance of the instance array.

The following example illustrates a design with bit-selected signal connecting to port of one instance. The width of the concatenation expression (4) is equal to the width of child port (1) multiplied by size of instance array (4). So, each bit/slice of a net in the concatenation expression connects with the port of one instance of the instance array.

```
# Design:
module test (
    output wrealsum out,
    input  wrealsum in,
    inout  wrealsum io
);
endmodule

module top ();
    wire net1, net2;
    wire net3, net4;
    wire net5, net6;
    wire [2:0]net7, net8, net9;

    test test[3:0] (.out({net1, net7[1:0], net2}),
                  .in ({net3, net8[2:1], net4}),
                  .io ({net5, net9[0], net9[2], net6}));
endmodule
```

Example 3

The following is an example of a part-selected signal connecting to a port of one instance. The width of concatenation expression (4) is equal to the width of child port (2) multiplied by the size of instance array (net 2). The concatenation expression separates into 2 parts. Each part with 2 bits/slices is connected with the port of one instance of the instance array.

```
# Design:
module test (
    output wrealsum out[1:0],
    input  wrealsum in[1:0],
    inout  wrealsum io[1:0]
);
endmodule

module top ();
    wire net1, net2;
    wire net3, net4;
    wire net5, net6;
    wire [2:0]net7, net8, net9;

    test test[1:0] (.out({net1, net7[1:0], net2}),
                  .in ({net3, net8[2:1], net4}),
                  .io ({net5, net9[0], net9[2], net6}));
endmodule
```

Example 4

If the width of concatenation expression of parent nets is equal to the width of individual child port, the tool connects the entire concatenation expression to the port of each instance in an array of instances.

The following is an example of how an entire concatenation expression connects to the port of each instance. The width of concatenation expression (3), equals to the width of child port (3); so, the entire concatenation expression is connected with the port of each instance of the instance array.

```
# Design:
module test (
    output EEnet out[2:0],
    input  EEnet in[2:0],
    inout  EEnet io[2:0]
);
endmodule
```



```
module top ();
    EEnet net1[2:0];
    EEnet net2[2:0];
    EEnet net3[2:0];
    EEnet net4[2:0];
    EEnet net5[2:0];
    EEnet net6[2:0];

    test test[1:0] (.out({net1[2:1], net2[0]}),
                   .in ({net3[2], net4[1:0]}),
                   .io ({net5[0], net6[0], net6[0]}));
endmodule
```

And, the tool reports an error in the following design scenarios:

- If a wire in concatenation expression is declared/resolved to electrical.
- If the nettype of concatenated parent wire is incompatible with that of child port in a design when an entire concatenation expression is connecting to each instance of instance array.
- If the nettype of concatenated parent wire is incompatible with that of child port in a design when the bit/part of concatenation expression connecting to an instance of instance array.
- If the concatenation expression contains non-collapsible net.

Limitations

Currently, the following connections are not supported:

- Mixed bus with electrical signal
- Multi-Dimensional arrays of nets in concatenation
- Multiple concatenation

Related Topic

- [Wreal Connections with Concatenated Wires and Nettype Ports](#)

Wreal Connections with Concatenated Wires and Nettype Ports

Connections between concatenated wires and compatible nettype port are supported by using the `-dms_concat` option. The nettype ports can be real valued net or struct SVUDN net. The tool applies the connection on RNM port of instance with concated wires, if the nettype is compatible.

And, the tool also enables RNM coercion of the nettype to parent wires in the concatenation expression, if the wires are coercible. However, for the such connections to function correctly, RNM processing must be enabled using the `-rnm_tech` option.

Currently, the following use cases are not supported:

- The connect module insertion between concatenated wires and RNM port
- Multiple concatenation

The following is an example of a connection with concatenated wires and nettype ports that are coercible. So, the `-rnm_tech` option is specified along with the `-dms_concat` option. The top module port, `top.a` is coerced to nettype `wreal` with resolution `CDS_res_wrealldriver` and `top.b` is coerced to nettype `wreal` with resolution `CDS_res_wrealldriver`. Also, two Wreal ports are declared: `top.test_b.a` is declared nettype `wreal` with resolution `CDS_res_wrealldriver` and `top.test_b.b` is declared nettype `wreal` with resolution `CDS_res_wrealldriver`.

```
xrun -clean top.sv -rnm_info -rnm_tech -dms_concat
```

```
-----
import cds_rnm_pkg::*;
nettype real wreal with CDS_res_wrealldriver;

module test ( a, b);
  inout a;
  inout b;

  wreal a;
  wreal b;

endmodule

module top (a, b);
  inout a;
  inout b;

  test test_b ( {b}, a);
```

endmodule

Related Topic

- [Wreal Connections with Concatenated Nettypes to Array of Instances](#)

SystemVerilog and AMS Extensions

You can simulate designs that contain SystemVerilog modules together with Verilog-AMS and VHDL-AMS modules. The following are the guidelines to follow in mixed-signal designs containing SystemVerilog and AMS modules:

- You may use either single-step simulation method to run a design with SystemVerilog and AMS blocks. When you use the `xrun` command, SystemVerilog and AMS parts of the design must be in separate files with appropriate file suffixes indicating design unit syntax (such as `.sv` for SystemVerilog and `.vams` for Verilog-AMS).
- Block Discipline Resolution (BDR) takes place whenever the `-sv` or `+sv` option is used on the `xrun` command line for any combination of Verilog, SystemVerilog or pure Verilog net connections.
- The SystemVerilog scope must not contain any explicitly declared electrical nets. However, nets in the SystemVerilog scope may become electrical using discipline resolution if connected to an electrical port.
- Any out-of-module reference (OOMR) to a SystemVerilog item from a Verilog-AMS or VHDL-AMS scope can only reference the types that can be used in hierarchical connections between the two languages.
- SystemVerilog and AMS blocks can be connected using the SystemVerilog real or logic variable data type in the following design configurations:
- **SystemVerilog on top instantiating Verilog-AMS:** In this configuration, a real or logic variable in the SystemVerilog scope can connect to an electrical, `wreal` port of the Verilog-AMS block below, or a SPICE port. If the electrical or SPICE ports are an output port, it is fully supported. However, if the lower connection is an input port or an inout port, then the simulator generates an error. See the [SystemVerilog on top instantiating Verilog-AMS: electrical port in Verilog-AMS](#) and [SystemVerilog on top instantiating Verilog-AMS: `wreal` port in Verilog-AMS](#) figures.

Figure 19.1: SystemVerilog on top instantiating Verilog-AMS: electrical port in Verilog-AMS

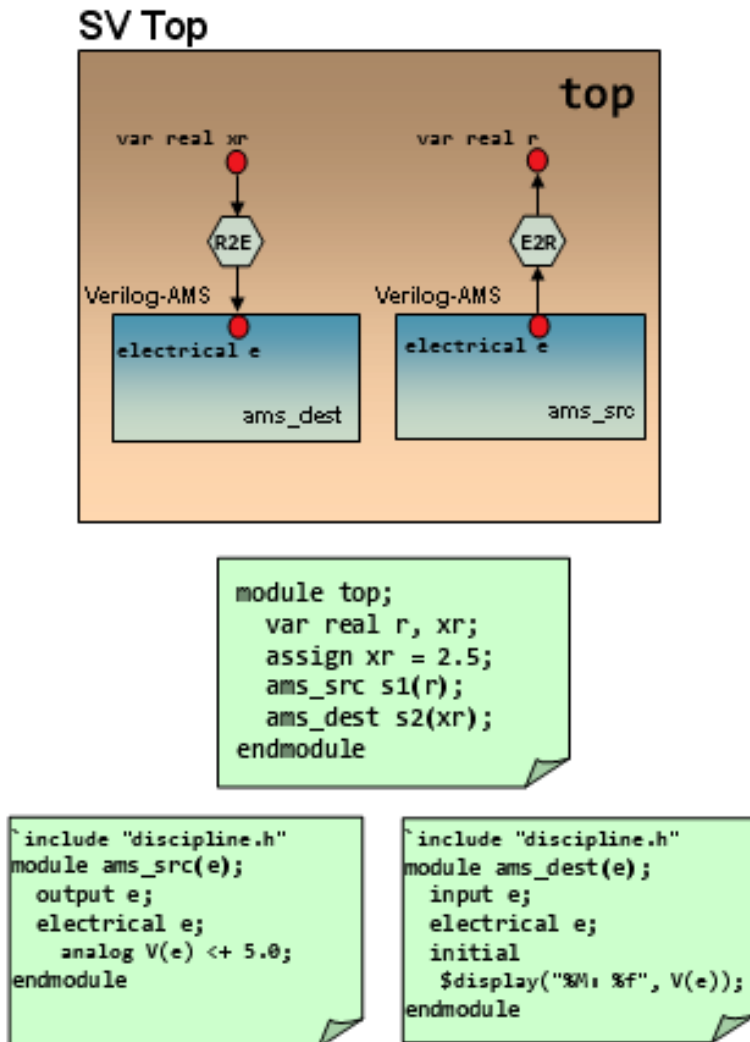
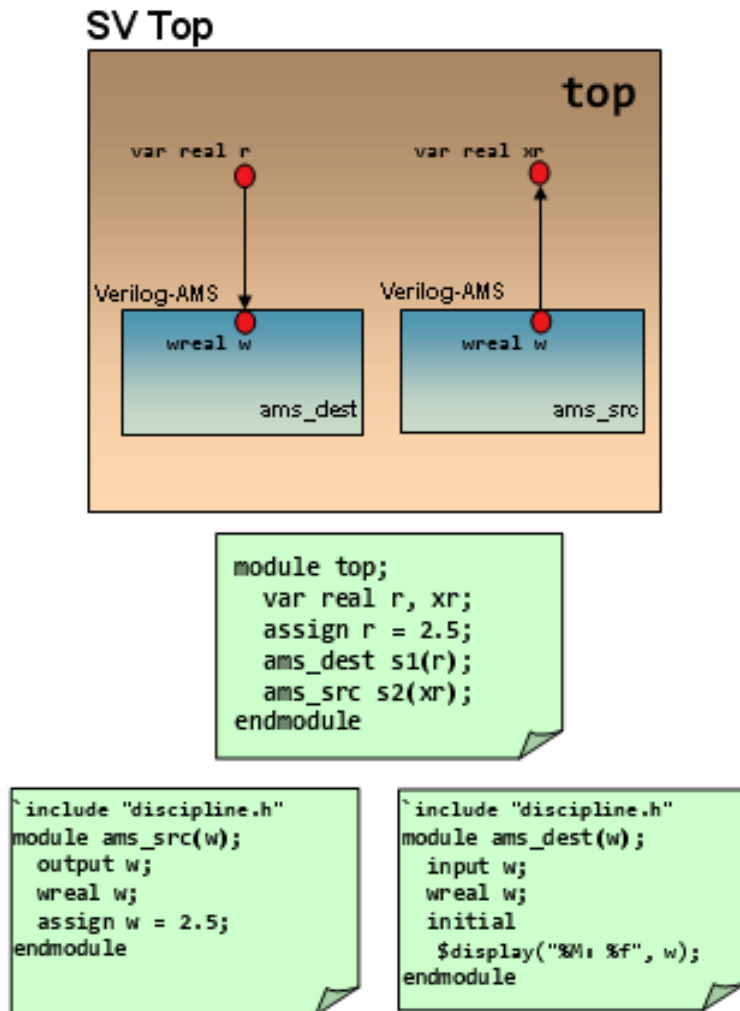


Figure 19.2: SystemVerilog on top instantiating Verilog-AMS: wreal port in Verilog-AMS



Power-smart Interface Elements (IEs) are supported to connect SystemVerilog real variable data type to Verilog or Verilog-AMS logic signal.

- **Verilog-AMS on top instantiating SystemVerilog:** In this configuration, an `electrical` or `wreal` port of the Verilog-AMS block can connect to a `real` variable in the SystemVerilog scope. If the port of the Verilog-AMS block is of `wreal` type, a connection is established with the SystemVerilog `real` variable without the need for any connection module (See the [Verilog-AMS on top instantiating SystemVerilog: wreal port in Verilog-AMS](#) figure).

If the port of the Verilog-AMS block is of `electrical` type, a connection is established with the SystemVerilog `real` variable by including an Electrical-To-Real connect module. (See

the [Verilog-AMS on top instantiating SystemVerilog: electrical port in Verilog-AMS](#) figure).

Figure 19.3: Verilog-AMS on top instantiating SystemVerilog: wreal port in Verilog-AMS

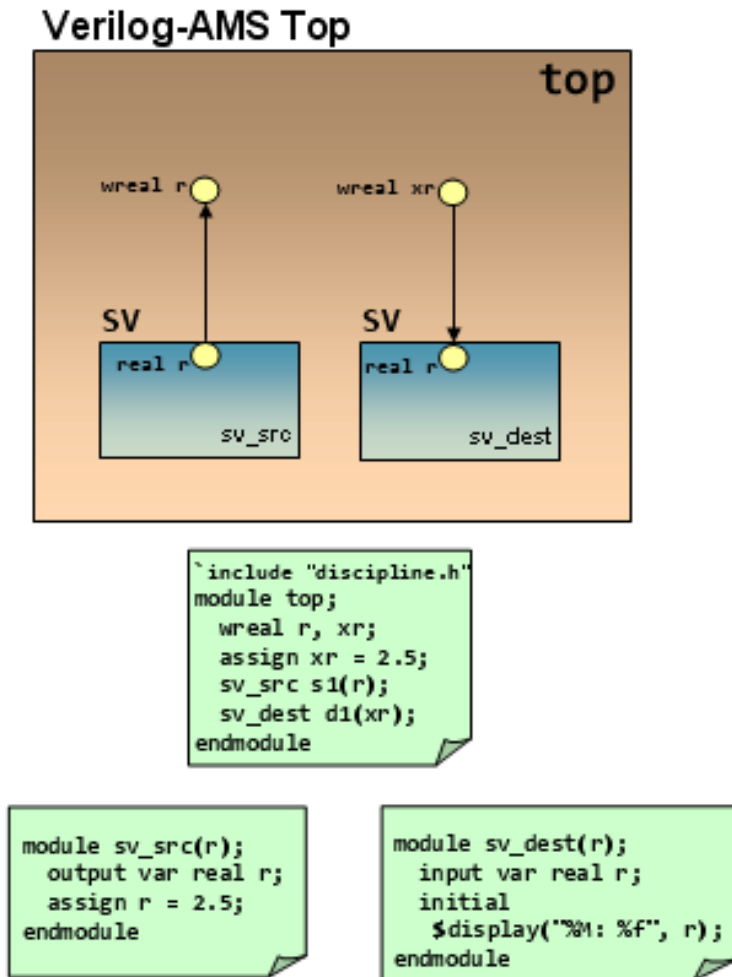
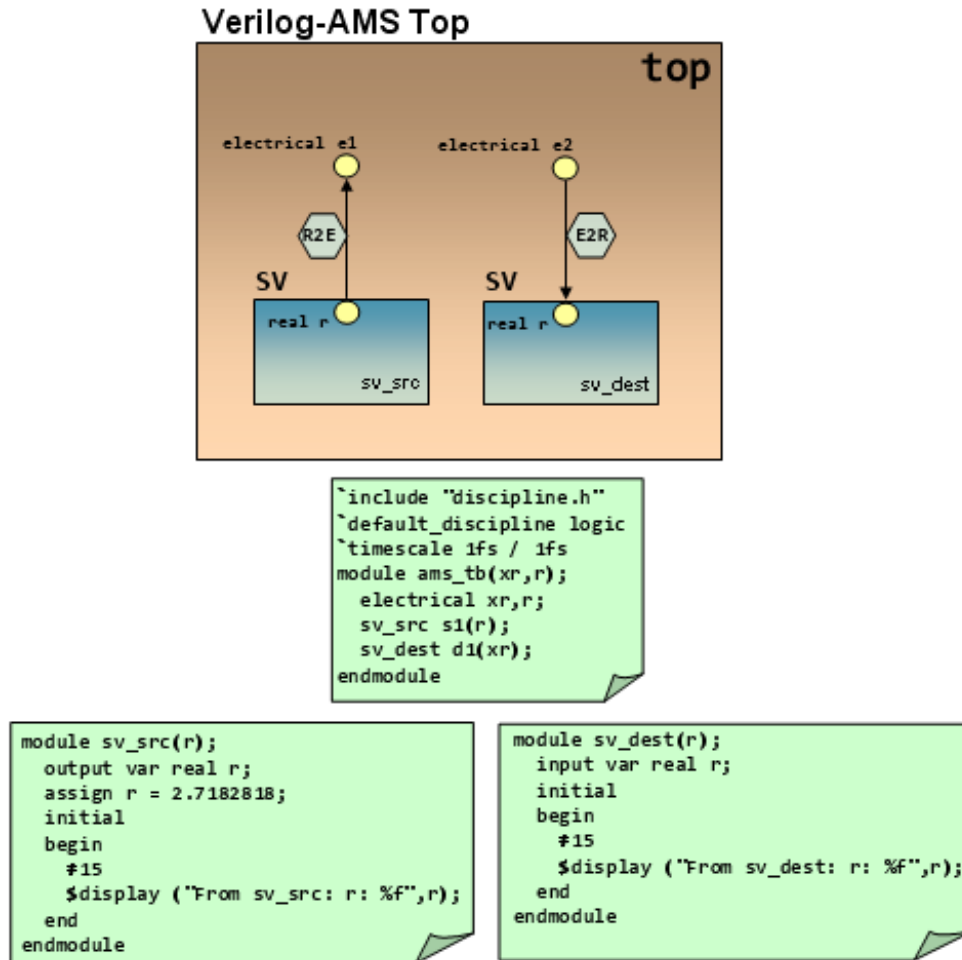


Figure 19.4: Verilog-AMS on top instantiating SystemVerilog: electrical port in Verilog-AMS



- Additionally, those hierarchical connections are allowed between SystemVerilog and Verilog-AMS, which are currently supported by the AMS Designer simulator between existing Verilog-2001 and Verilog-AMS data objects.

The following SPICE units are not recognized by SystemVerilog:

- Built-in primitives such as resistor, capacitor, or [MOSFET](#)
- Primitives brought in through [CMI](#) libraries
- Primitives brought in using the `MODELPATH` option

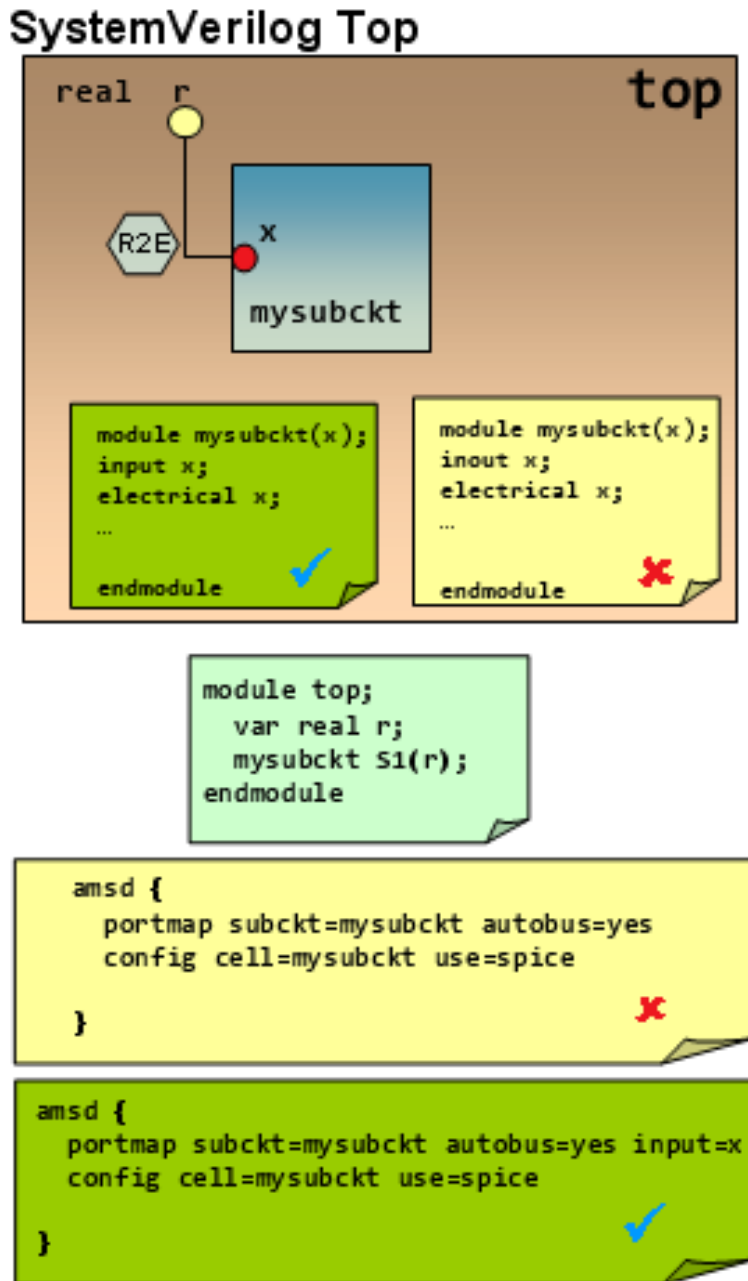
If the tool encounters such a primitive, it searches for a regular master of the same name; and when the master is not found, the elaboration step exits with an error.

- A SPICE subckt can be instantiated in SystemVerilog with restriction if the connection is made using the SystemVerilog real variable. In such a scenario, the direction of each port of the subckt must be clearly specified as either input or output by using any of the following

three ways:

- By using the `reffile` option on the AMS control block `portmap` card. This points back to the original Verilog-AMS file from which the port directions are taken.
- By using an `input` or `output` option on the port in question on the AMS control block `portmap` card.
- By using a `file` option for a port bind file specified on the AMS control block `portmap` card. This port bind file contains explicit instructions for mapping the SPICE ports to the ports in the generated skeleton. These instructions can include port directions. Each port of the subckt must be clearly designated. The reason for this restriction is that the SystemVerilog real variable can only support a single driver, and therefore cannot be connected to bidirectional (inout) ports. If the above conditions are not met, an error will be issued stating that connection of a real variable to the inout port of a SPICE instance is not supported. See the [SystemVerilog on top, SPICE subckt Underneath](#) figure.

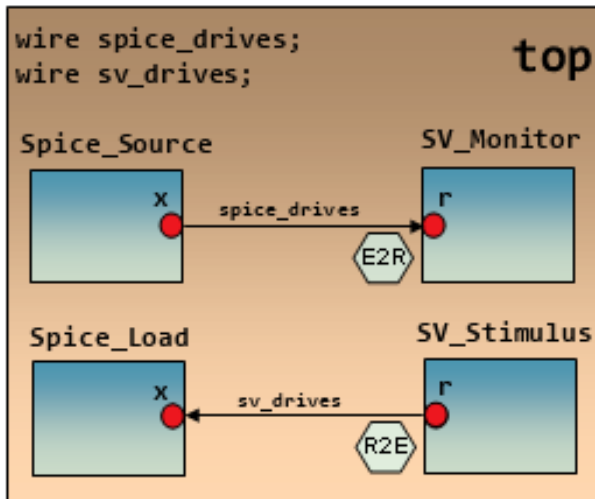
Figure 19.5: SystemVerilog on top, SPICE subckt Underneath



- Side-by-side connection between SPICE and SystemVerilog blocks is allowed to occur in Verilog or SystemVerilog scope, regardless of whether the SystemVerilog block or the SPICE block acts as the driver. See the [Side-by-side connection between SPICE and SystemVerilog blocks](#) figure.

Figure 19.6: Side-by-side connection between SPICE and SystemVerilog blocks

Top (Verilog or SV)



```

module top;
  wire spice_drives;
  wire sv_drives;
  Spice_Source V1(spice_drives);
  SV_Monitor M1(spice_drives);
  Spice_Load L1(sv_drives);
  SV_Stimulus S1(sv_drives);
endmodule
  
```

```

subckt Spice_Source(x)
  V1 (x 0) vsource type=pulse ...
  ...
ends Spice_Source
  
```

```

module SV_Monitor(x);
  input var real x;
  initial $display("x=%g", x);
endmodule
  
```

```

subckt Spice_Load(x)
  R1 (x 0) resistor r=10K
  ...
ends Spice_Load
  
```

```

module SV_Stimulus(x);
  output var real x;
  initial x = 2.5;
endmodule
  
```

Related Topics

- [SystemVerilog Binding on SPICE](#)
- [Wildcard-Named Port Connections in AMS](#)
- [Connecting SystemVerilog Interface to DMS/Verilog-AMS \(Electrical\) Signals](#)
- [Coercion for SystemVerilog Interfaces, Program, or Checker Blocks](#)
- [Using Custom ie Parameter and Connect Module Names for UDN-to-Electrical Connections](#)
- [Configuring SV-AMS Connect Modules for UDN-UDN, UDN-Logic, and UDN-Real Connections](#)
- [Using the SVAMS Parser](#)

SystemVerilog Binding on SPICE

Spectre AMS Designer supports the `SV bind` statement that enables you to specify one or more instantiations of a module without modifying the code of the target. For example, you can use the `SV bind` statement at SPICE level to access the SPICE nodes and perform verification, such as adding assertion on them. For example:

```
//cell target

bind analog_top inc #(param_list) xinc (port map);

// instance target
bind top.i1 inc #(param_list) xinc (port map);
```

You can use the `SV bind` statement in a cell or instance specification.

In the above example, if cell `analog_top` or instance `top.i1` is a SPICE subcircuit that is configured through the `amsd` control block, it triggers SV binding on SPICE.

SV binding targets the top-level SPICE instance block as well as any other instance within the scope of the hierarchy of the top-level SPICE instance block; however, there is a restriction to targeting SPICE instances below the top-level SPICE instance block. You need to ensure that the SPICE subcircuit that is instantiated within the top-level SPICE hierarchy is also instantiated in the digital instance. For example:

```
// Here, top.ana_top_inst of subckt ana_top is verilog-spice boundary;
ana_top_level2 is an instance inside ana_top

bind top.ana_top_inst.ana_top_level2_inst inc #(param_list) xinc (port map);
```

```
//analog.scs
.subckt ana_top
ana_top_level2_inst ana_top_level2 a b
.endckt

//amscf.scs

amsd{
portmap subckt=ana_top
config cell=ana_top use=spice
portmap subckt=ana_top_level2 //This is required
config cell=ana_top use=spice //This is required
}
```

SV `bind` statement does not support the dot star (.) notation.

You can ignore the SV bind statements on SPICE blocks by using the `xrun` option `-ignore_svbind_spice`.

Related Topics

- [Wildcard-Named Port Connections in AMS](#)
- [SystemVerilog and AMS Extensions](#)

Wildcard-Named Port Connections in AMS

In SystemVerilog, you can use the wildcard-named port connections, which are represented using the dot-star notation (.*), to specify port connections in the mixed-signal flow. It provides a concise and powerful implicit port connection and avoids the top-level verbosity to explicitly detail the port connection for each instance.

Currently, the following are supported:

- Ports that are basic Verilog types, Verilog-AMS signals, or a combination of both
- Ports that are of user-defined nettype (UDN). Currently, only those port connections where both the port expression and the formal port are UDNs that have structure as their user-defined type are supported.
- Supported UDN-to-UDN connections include:
 - explicit or implicit interconnects or wires coerced to UDNs

- scalars
- arrays
- Unsupported scenarios for all connections as explicitly named port connections
 - Usage in mixed-signal context
 - Complete list of port signals connected using a single `.*`.
 - Arrays and vectors

Wildcard-named port connections of UDNs with UDTs and UDRs are supported using explicit port connections.

The following do not support the `.*` wildcard construct:

- Complex SV data types including enum, union, class, and interface class
- Generic wraels on SV side
- SimVision

Following is an example of wildcard-named port connections:

```
test.sv:
`timescale 1ns/1ps
module top;
    wire m,n;
    m_ams ams(.*);
    assign m = 1;
endmodule
module m_sv(input logic m, output logic n);
endmodule

test.vams:
`include "disciplines.vams"
`timescale 1ns/1ps
module m_ams(m,n);
    input m;
    wire m;
    output n;
    electrical n;
m_sv sv(.m(m),.n(n));
analog begin
    V(n) <+ 1.8;
end
endmodule
```

The following example illustrates the use of arrays and vectors as port connections that are connected using wildcard-named port connection.

```
tb.sv:
module tb();
    reg l1l;
    reg [3:0] mmm;
    dut dut_inst(. *);
endmodule

dut.v:
module dut (l1l, mmm);
    input l1l;
    input [3:0] mmm;
    spicex myspicex (.l1l(l1l), .mmm(mmm));
endmodule

spice.scs:
simulator lang=spectre
    subckt spicex l1l mmm_3 mmm_2 mmm_1 mmm_0
ends spicex

spice.v:
module spicex (input l1l, input [3:0] mmm);
endmodule

amscf.scs:
simulator lang=spectre
include "spice.scs"
tran tran stop=1u
amsd{

portmap subckt=spicex porttype=name reffile="spice.v"
    config cell=spicex use=spice
    ie vsup=1.8
}

xrun tb.sv dut.v spice.v amscf.scs -timescale 1ns/1ps
```

The following example illustrates how to connect a UDN port using wildcard-named port connection:

```
//Custom package

package myPkg;
typedef struct{

    real x;
```

```
}myStruct;

function automatic myStruct res_myStruct(input myStruct driver[]);

    foreach(driver[i])
        res_myStruct.x += driver[i].x;

endfunction

nettype myStruct mySum with res_myStruct;
endpackage

//import native packages

import cds_rnm_pkg::*;

import EE_pkg::*;

//import custom packages
import myPkg::*;

//.* connected testbench
module top;
    interconnect x,y;
    interconnect u,v;

    test1 u0(.*);
    test2 u1(.*);

endmodule // top

//Module Definitions
module test1(x,y,u,v);

    input wrealldriver x;
    input mySum y;
    inout EEenet u;
    inout wire v;

endmodule // test

module test2(x,y,u,v);

    output wrealldriver x;
    inout mySum y;
    inout EEenet u;
    inout wire v;

endmodule // test
```


Related Topic

- [SystemVerilog and AMS Extensions](#)

SystemVerilog Assertions on real, wreal, and electrical Nets

The AMS Designer simulator and Xcelium simulator with mixed-signal option let you use SystemVerilog Assertions on SystemVerilog `real` variables (or ports) that connect to `wreal` or `electrical` nets, and PSL assertions on `real` and Verilog-AMS `wreal` nets that connect to `electrical` nets.

PSL stands for the property specification language that is undergoing standardization as IEEE 1850 Property Specification Language.

SystemVerilog Assertions

In mixed-signal designs, you can specify a SystemVerilog Assertion (SVA) on a SystemVerilog `real` port that connects to a `wreal` or `electrical` net. You can use the SVA to check the value of the `wreal` or `electrical` net.

When you connect a SystemVerilog `real` port to an `electrical` net, the elaborator inserts the appropriate real-to-electrical or electrical-to-real interface element automatically.

In the following example, SystemVerilog `real` number ports (`r`, `xr`, and `wr`) are connected to nets belonging to the electrical domain and nets of type `wreal`.

```
module top;

    var real r, xr, wr;

    assign xr = 3.14;

    ams_electrical_src e_sl(r); // causes insertion of Electrical2Real connection
                               // module

    ams_electrical_dst e_dl(xr); // causes insertion of Real2Electrical connection
                               // module
```

```
    ams_wreal_src      w_sl(wr); // Connects SystemVerilog real variable to wreal
endmodule

module ams_electrical_dst(e);

    input e; electrical e;
    initial #10 $display("%M: %f", V(e));

endmodule

module ams_electrical_src(e);

    output e; electrical e;
    analog V(e) <+ 5.0;

endmodule

module ams_wreal_src(w);

    output w; wreal w;
    assign w = 2.5;

endmodule
```

Once the SystemVerilog real variables import the AMS functionality through the code above, they can appear in any SystemVerilog assertion statement that permits the use of real variables. `real` variables can be used in SystemVerilog assertion, `real` variable, `r`, that comes in as a `wreal` port (from the `ams_design` instance), `w`:

```
module sv_tb;
    real r;
    time t;

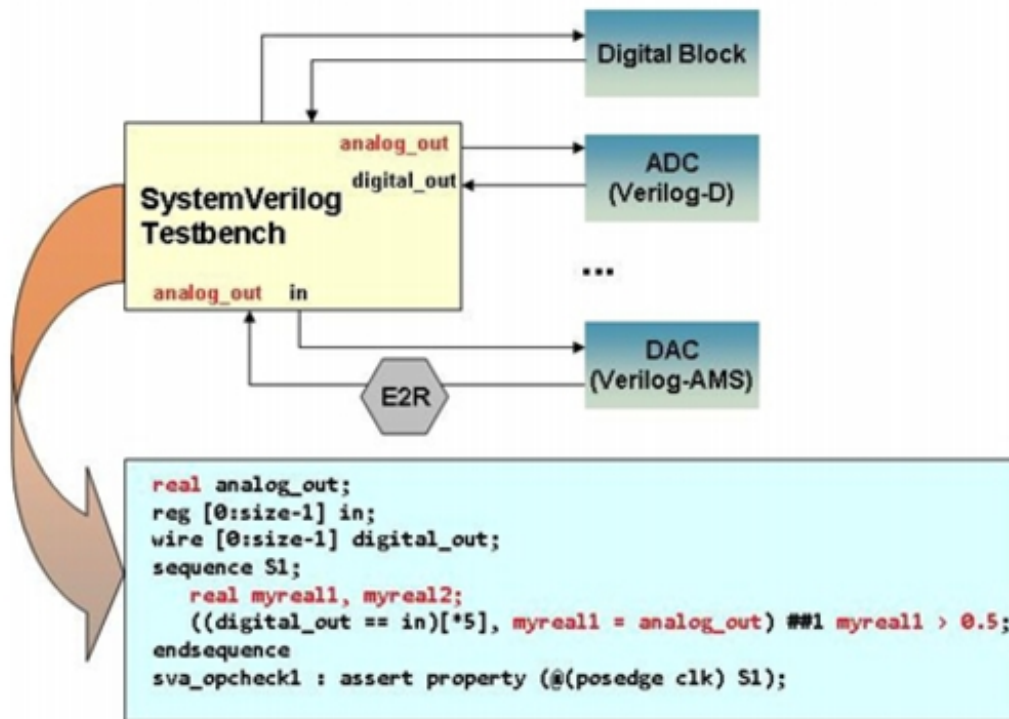
    ams_design a1(r);

    always @(r) begin
        assert (r < 5.5) else begin
            t = $time;
            $error("assert failed at time %0t",t);
        end
    end
endmodule

module ams_design(w);
    output w; wreal w;
    assign w =5.6;
end
```

The [SystemVerilog real variable used inside a SystemVerilog assertion](#) figure shows another example of a SystemVerilog real variable being used inside a SystemVerilog assertion.

Figure 19.1: SystemVerilog real variable used inside a SystemVerilog assertion



Analog System Tasks in SVA

SystemVerilog Assertions (SVA) can contain analog system tasks like `$cds_get_analog_value()` to access the values of analog objects for use within assertions. SVA semantics specify the use of sampled values of all signals, except local variables, during assertion evaluation.

However, if any digital signal changes during assertion evaluation, there is the risk of sampling analog objects (via `$cds_get_analog_value()`) too. This is not acceptable because values of such analog objects could have changed significantly since the last sampling and may be out of date for a fair evaluation.

Consider the following example:

```

default clocking CLK @(negedge clk); endclocking

assert property (
    a & !b
    |->
    $cds_get_analog_value("top.simpleVAMS.x") > c
    )
    
```

```
);
```

The signals of the assertion above would be sampled whenever digital signals, 'a', 'b', 'c', or 'clk' change, whereas the assertion would be evaluated at the negative edge of 'clk'.

So the value of the expression `$cds_get_analog_value("top.simpleVAMS.x") > c` used in the assertion evaluation would be the value that existed when the signals were sampled the last time.

To eliminate the evaluation error caused by this, the assertion evaluation process treats the expression just like local variables and uses the current value of the expression

```
$cds_get_analog_value("top.simpleVAMS.x") > c
```

Note, however, that like local variables, it uses the current value of ALL digital signals in the expression that contains the analog system function. In this case, the current value of 'c' would be used.

This can cause race conditions when such digital signals change on the same edge as the clock. This is currently a limitation in the general assertion solution.

Related Topics

- [SystemVerilog and AMS Extensions](#)
- [SVA Quick Reference](#)

Using Custom ie Parameter and Connect Module Names for UDN-to-Electrical Connections

In addition to the built-in `ie` parameter and connect module names, you can also provide custom names for the parameters and connect modules that are defined in the UDN connect modules within an `ie` card statement.

To enable custom parameter names and connect module names, you must use the `-custom_udn_cr` option in the `xrun` command line.

You must use the `ie` card statement in the `amsd` block to specify the parameter name of your choice. The tool automatically collects the valid parameter set. For example, in the following code, the custom parameters defined are `my_vtol`, `my_itol`, and `custom_vddl0`.

```
amsd {  
  ie vsup=3.0 my_vtol=1e-5 ttol=1e-6 vddl0=15 nettype="myNT"  
  ie vsup=3.0 my_vtol=1e-1 my_itol=1e-6 custom_vddl0=10
```

```
    inst=top.dig_driver
nettype="myNT"
}
```

Similarly, you can set any name for the connect module. The tool internally sets up the mapping between the nettype and its custom modules. For example, in the following code the custom module name defined is `nt2e`.

```
// nt2e.svams
`include "disciplines.vams"
import udn_package::*;
connectmodule nt2e( Din, Aout); //input myNT Din, output electrical Aout);
    input Din;
    myNT Din;
    output Aout;
    electrical Aout;
```

The following example illustrates the usage of custom parameter names and connect module names for Electrical-to-UDN, UDN-to-Electrical, and bidirectional connections.

```
// udn_package
package udn_package;
    typedef struct {
        real r;
    } myT;
    nettype myT myNT with RESFUNC;
endpackage

// top.sv
import udn_package::*;
module top;
    myNT e2udn;
    myNT udn2e;

    //Electrical to UDN
    elect_sine_driver xelect_sine_driver (e2udn);
    sine_mon xsine_mon1(e2udn);

    //UDN to Electrical
    digital_nt_driver dig_driver (udn2e);
    analog_electrical_sink analog_load (udn2e);
endmodule

//sine Monitor
module sine_mon(nt1);
    input nt1;
```

```
    myNT nt1;
always @(nt1.r) begin
$display("electrical to sine:%0f",nt1.r);
end
endmodule

//digital driver
module digital_nt_driver (out);
    output out;
    myNT out;
    assign out = '{1.5};
endmodule

// analog.vams
`include "disciplines.vams"
module elect_sine_driver (sigout);
    output sigout;
    electrical sigout;
    electrical gnd;
    ground gnd;

vsource #(.type("sine"), .ampl(5.0), .dc(5.0), .freq(1.0M)) v0 (sigout, gnd);
endmodule

module analog_electrical_sink(in);
    input in;
    electrical in, out;
    real Dreg_in;

    always @ (absdelta(V(in), 0.001, 1n, 0.001/4))
    begin
        Dreg_in = V(in);
    end
endmodule

//You can define the custom parameter and connect module names in the following amsd
block.

amsd {
    ie vsup=3.0 my_vtol=1e-5 ttol=1e-6 vdd10=15 nettype="myNT"
    ie vsup=3.0 my_vtol=1e-1 my_itol=1e-6 custom_vdd10=10
    inst=top.dig_driver
nettype="myNT"
}

//Electrical-to-UDN connections. A custom parameter "my_vtol" and a custom connect
module name "e2myNT" are defined in this code.
```

```
`include "disciplines.vams"
import udn_package::*;
connectmodule e2myNT ( Ain, Dout); // input electrical Ain, output      myNT Dout
    input Ain;
    electrical Ain;
    output Dout;
    myNT Dout;
    parameter real vss=0 ; //from (-inf:inf); //Voltage of negative supply
    parameter real vsup=5.0 ;//from (0:inf); //Supply voltage based on vss
    parameter real vdelta=vsup/64 ; //from (0:inf); //voltage delta
    parameter real my_vtol=vdelta/4 ; //from (0:vdelta); //voltage tolerance
    parameter real vdd=vss+vsup; //internal parameter: vss+vsup
    parameter real vddl0=vdd+vsup*10; //internal parameter: vdd+vsup*10
    parameter real vssl0=vss-vsup*10; //internal parameter: vss-vsup*10

-----
-----
endconnectmodule

//UDN-to-electrical connections. Three custom parameters "my_vtol", "my_itol", and
"custom_vddl0"; and, a custom connect module name "nt2e" are defined in the following
part.

import udn_package::*;
connectmodule nt2e( Din, Aout); //input myNT Din, output electrical Aout);
    input Din;
    myNT Din;
    output Aout;
    electrical Aout;

    parameter real vdelta=vsup/64;    // voltage delta
    parameter real vx=vss;    // X output voltage
    parameter real my_vtol=vdelta/4; // voltage tolerance
    parameter real my_itol=idelta/4; // current tolerance
    parameter real vdd=vss+vsup;      // internal parameter: vss+vsup
    parameter real custom_vddl0=vdd+vsup*10; // internal parameter: vdd+vsup*10
    parameter real vssl0=vss-vsup*10;    // internal parameter: vss-vsup*10

-----
-----
endconnectmodule
```

Related Topics

- [Configuring SV-AMS Connect Modules for UDN-UDN, UDN-Logic, and UDN-Real Connections](#)
- [Complex Port Definitions with User-Defined Nettypes](#)

Configuring SV-AMS Connect Modules for UDN-UDN, UDN-Logic, and UDN-Real Connections

You can enable automatic insertion of SV-AMS connect modules for mixed-signal designs where the nettype is one of the following:

- A wire that is coerced
- A wreal type
- A User-Defined Net (UDN)
- An array port that is connected by a concatenated collection

The direction of the connect module is determined by the direction of the port that is declared across which it is inserted (input, output, or inout). Both the highcon and lowcon of the port, must be nets and the port must be collapsible for a connect module to be inserted.

At elaboration time, when the tool detects port connections, where both the upper and lower connected nets have different nettype, it inserts the custom SV-AMS connect modules with the correct parameterization.

Currently, the tool supports the following types of connect modules:

- UDN to UDN
- UDN to Logic
- UDN to Real

To insert custom SV-AMS connect modules, specify the `-rnm_dmsie` option, which instructs the tool to insert the SVAMS connect modules where needed. The `-rnm_dmsie` option must be specified along with either of the following options:

- `-custom_udn_cr`: Instructs the tool to generate custom SVAMS connect rules
- `-auto_config_svams_ie`: Instructs the tool to generate connect modules automatically

You must also specify the other required files such as the `ams_cf.scs` file, the SystemVerilog file

(.sv), where the SystemVerilog block/structural netlists are defined, and the SystemVerilog-AMS (.svams) file that contains the connect module descriptions.

The following table lists the use models currently supported for analog mixed-signal designs or DMS SV-RNM designs.

Using the - custom_udn_cr option	Use Model 1	For analog mixed-signal designs or DMS SV-RNM designs with an ie card statement where you want to specify: • Custom v_{sup} value • Custom UDNs
	Use Model 2	For DMS SV-RNM designs with an existing ie card statement where you want to specify: • The default v_{sup} value, 1.8 • Custom UDNs
	Use Model 3	For DMS SV-RNM designs with an ie card statement where you want to specify: • Custom v_{sp} value • Custom UDNs • Other parameters such as t_{delay} or cell This allows you to have a high level of control on IE configuration.
Using the - auto_config_svams_ie option	Use Model 4	For DMS SV-RNM designs that do not use ie card and amsd block, and do not have a spectre *.scs file. Then, the tool automatically creates an ie card with default v _{sup} (1.8) and creates all SV UDN nettype list/connect module list based on the connect module that you have provided.

Use Model 5	For DMS SV-RNM designs that use an <code>ie</code> card and you want to specify only the <code>vsup</code> value. The tool uses the specified <code>vsup</code> value and creates all SV UDN nettype list/connect module list based on the connect module that you have provided.
Use Model 6	For designs where the <code>highcon</code> expression is a concatenation expression and you want the tool to insert a connect module automatically. In this case, you must specify - <code>rnm_dmsie</code> and <code>-dms_concat</code> options along with the <code>-auto_config_svams_ie</code> to enable the connection between concatenated wires and compatible nettype.

Limitations of SV-AMS Connect Modules

- SV-AMS connect modules are not inserted for port connections of variable ports, constant expressions, constants, arithmetic expressions, and non-constant array selects.
- When using multiple wreal types you need a set of SV-AMS connect modules for each wreal type.
- SV-AMS connect modules inserted on discrete nets that are also connected to SIE module ports (like a wreal tran) are not supported and results in an error. You encounter the `*E, SIEDTM` error if the connect module is inserted directly on the SIE port and if the connect module is inserted further up in the hierarchy the `*E, B2BSIE` error is generated.
- Insertion of connect modules at mixed language boundaries such as VHDL and SystemC.
- All current restrictions on R2L IEs remain as restrictions on DMS IEs as well.
- IE optimization is not applied to these connect module instances.
- `ie` card with the `connrules` parameter specified is not supported.
- SV-AMS connect module insertion is not supported with Incremental Elaboration.

Use Model 1

For analog mixed-signal designs or DMS SV-RNM designs that have an `amsd` control block defined. You must use the `ie` card statement in the `amsd` block to specify the `vsup` value, `nettype` parameter names of your choice. The `nettype` parameter should have all valid UDN types used in design which needs SVAMS connect module insertion.

The `amscf.scs` file is then passed on to `xrun`, along with the `-rnm_dmsie` and the `-custom_udn_cr` options. And, the tool automatically collects the valid parameter set from the provided IE card.

Syntax:

```
% xrun -custom_udn_cr -rnm_dmsie amsd.scs connect_modules.svams top.sv
```

Here is an example of the `amsd` control block where a custom SV-UDN, `myUDN` is introduced using the `nettype` parameter along with the `vsp` value:

```
//amsd.scs file
amsd {
    ie vsup=2.5 nettype="myUDN"
}
```

Use Model 2

In this use model an existing amsd control block can be used for SV-AMS connect module by simply adding only the `nettype` parameter. And, the tool implicitly creates an `ie` card with the default `vsup` value, 1.8 and creates the connect rules based on the connect module that you have provided.

Syntax:

```
% xrun -custom_udn_cr -rnm_dmsie amsd.scs connect_modules.svams top.sv
```

Here is an example of the amsd control block where custom SV-UDNs, `nettypea` and `nettypeb` are introduced using the `nettype` parameter:

```
//amsd.scs
amsd{
    ie nettype="nettypea nettypeb"
}

//Or:

amsd{
    udt nettype="nettypea nettypeb"
}
```

Use Model 3

In this use model, `vsup` is supported within the custom SV-AMS connect modules along with various other parameters that allows you to have a high level of control on IE configuration. `ie` card can parameterize these SV-AMS connect modules within various scopes.

Syntax:

```
% xrun -custom_udn_cr -rnm_dmsie amsd.scs connect_modules.svams top.sv
```

Here is an example where the parameters `vsup` and `tdelay` are parameterized within a different cell scope

```
//amsd.scs
amsd {
    ie vsup=1.8 nettype="myUDN"
    ie vsup=4.0 tdelay=2.5 cell=ana_top_1 nettype="myUDN"
}
```

Use Model 4

For designs that do not use `ie` card and `amsd` block, and do not have a `spectre *.scs` file, the simplest way to integrate SV-AMS connectmodules is by specifying the `-auto_config_svams_ie` and `-rnm_dmsie` options along with the source files, and SV-AMS connect modules.

Then, the tool automatically creates `ie` card with default `vsup=1.8` and create all SV UDN nettype list/connect module list based on the connect module that you have provided. In this use model, it is assumed that there are no logic connections to any other type such as logic-to- real, logic-to-electrical in the design.

Syntax:

```
% xrun -auto_config_svams_ie -rnm_dmsie connect_modules.svams top.sv
```

Use Model 5

For DMS SV-RNM designs that use `ie` card when you want to specify a `vsup` value. The tool uses the specified `vsup` value and creates all SV UDN nettype list/connect module list based on the connect module that you have provided.

Syntax:

```
% xrun -auto_config_svams_ie -rnm_dmsie connect_modules.svams top.sv
amsd{
ie vsup=2.5
}
```

Use Model 6

For designs where the `highcon` expression is a concatenation expression and you want the tool to insert a connect module automatically, you must specify the `-auto_config_svams_ie`, `-rnm_dmsie`, and `-dms_concat` options along with the source files, and user-defined SV-AMS connect modules.

Syntax:

```
% xrun -auto_config_svams_ie -rnm_dmsie -dms_concat connect_modules.svams top.sv
```

The tool inserts a connect module automatically for each bit or slice of concatenated wire and port

connection. The nets in concatenation expression could be scalar, single dimensioned unpacked array or bit/slice/part select of that net. The concatenated wires could have different data type or nettype such as electrical, Wreal, SV-Wreal, and SV-UDN.

For example, in the following design, the nets in concatenation expression have different nettypes. The connect module L2EEnet/EEnet2L is inserted on the slices of ports between logic wire and SV-UDN (EEnet) and the connect module R2EEnet/EEnet2R is inserted on the slices of ports between SV wreal and SV-UDN (EEnet).

```
import cds_rnm_pkg::*;
import EE_pkg::*;
nettype wrealsum mywreal ;

module top();
mywreal drv_r[2:0];
wire [2:0]drv_l;
EEnet drv_e[2:0];
mywreal rcv_r[2:0];
wire [2:0]rcv_l;
EEnet rcv_e[2:0];

child inst(
.in ({drv_r[2],drv_l[2],drv_e[2],drv_r[1:0],drv_l[1:0],drv_e[1:0],drv_r,drv_l,drv_e}),
.out ({rcv_r[0],rcv_l[0],rcv_e[0],rcv_r[2:1],rcv_l[2:1],rcv_e[2:1],rcv_r,rcv_l,rcv_e})
);
endmodule

module child(in, out);
input [17:0]in;
output [17:0]out;
EEnet in[17:0];
EEnet out[17:0];
Endmodule=====IE info=====
```

```
IE Report Summary (with disciplines and directions):
Thev2Nort ( ddiscrete_1_8 input; ddiscrete_1_8 output;) total: 5
Nort2Thev ( ddiscrete_1_8 output; ddiscrete_1_8 input;) total: 5
-----
```

Effective Number of IE Instances:
Total Number of Connect Modules : 10

Related Topic

- [Using Custom ie Parameter and Connect Module Names for UDN-to-Electrical Connections](#)

Complex Port Definitions with User-Defined Nettypes

You can have complex port connections with System Verilog user-defined ntypes (UDN) in your design. All the elements inside complex ports need to be declared as wire or interconnect. This allows the tool to coerce the complex port to the appropriate UDN type. The direction of the port can be input, output, or inout.

Port expressions can be any of the following:

- Bit-select of UDN array used in port concatenation. For example:

```
module dut ( .aain({aain[1], aain[0]}) );  
    input [2:0] aain;  
    cell1 I0 ( .ain(aain));  
endmodule
```

- Part-select of UDN array used in port concatenation. For example:

```
module dut ( .aain({aain[1:0], aain[3:2]}) );  
    input [3:0] aain;  
    cell1 I0 ( .ain(aain));  
endmodule
```

- Identifier of UDN used in port concatenation. For example:

```
module dut ( .aain({a3, a2}) );  
    input a2, a3;  
    wire a1;  
    cell1 I0 ( .ain({a3, a2, a1}));  
endmodule
```

- IPart-select of UDN array used in port definition (no concatenation)

```
module dut ( .aain(aain[1+:2]) );  
    input [2:0] aain;
```

```
cell1 I0 ( .ain(aain));
endmodule
```

- **Combination of above used in port concatenation.** For example:

```
module dut ( .aain({aain[1], aain[3:2]}) );
    input [3:0] aain;
    cell1 I0 ( .ain(aain));
endmodule
```

- **Bit-select of UDN array used in port definition (no concatenation).** For example:

```
module dut ( .aain(aain[0]) );
    input [2:0] aain;
    cell1 I0 ( .ain(aain));
endmodule
```

- **Part-select of UDN array used in port definition (no concatenation).** For example:

```
module dut ( .aain(aain[1:0]) );
    input [2:0] aain;
    cell1 I0 ( .ain(aain));
endmodule
```

The following is an example of a complex port definition:

```
typedef struct {
    real V;
    real I;
} real_wire_t;

function automatic real_wire_t real_wire_res(input real_wire_t driver[]);
    foreach (driver[i]) begin
        real_wire_res.V += driver[i].V;
        real_wire_res.I += driver[i].I;
    end
endfunction

nettype real_wire_t real_wire with real_wire_res;

module top ( );
    wire [1:0] aain;
    dut I0 ( .aain(aain));
endmodule

module cell1(ain);
    input real_wire ain[2:0];
    assign ain[0] = '{1, 1.1};
    assign ain[1] = '{2.2, 2.1};
endmodule
```


Using the SVAMS Parser

System Verilog and Verilog-AMS constructs can be used with the SystemVerilog AMS (SVAMS) Parser, which is now integrated with the native verilog (*xmvlog*) parser. This means that UDN-to-UDN, UDN-to-logic, UDN-to-electrical, UDN-to-Wreal connect modules can be parsed using the *xmvlog* parser. Dynamic Voltage Supply (DVS) features (the `$real_net_alias` system task) can also be parsed using the *xmvlog* parser.

This allows you to create SVAMS connect modules with a more extended set of SystemVerilog and Cadence language constructs.

The following are the unsupported use cases of SVAMS parser:

- Verilog-AMS logic, Verilog-AMS wreal
- Connections with variable ports such as real vars, structs types
- Connections with datatypes UDT-Struct, union, enum
- Generation block inside SVAMS connect modules
- System Verilog assertions and PSL's assertions are not allowed
- SystemVerilog testbench constructs such as interface, program, checker, modports, clocking blocks, module/endmodule
- Using the Verilog-AMS `table_model` function is not supported
- SVAMS constructs such as `initial_step`, `final_step`, and `bound_step` are not supported for UDN-to-logic, UDN-to-real, and UDN-to-UDN connections
- Using analog constructs (electrical) in UDN-UDN, UDN-logic, UDN-wreal connect modules
- Importing DPI functions

Related Topics

- [Configuring SV-AMS Connect Modules for UDN-UDN, UDN-Logic, and UDN-Real Connections](#)
- [\\$real_net_alias](#)

Variable To Wire (VTW) Optimization in Mixed-Signal Designs

For designs having mixed signal nets, you can enable Variable to Wire (VTW) optimization using the `-dms_perf` option.

The tool analyzes whether a logic or reg data type can be safely treated as a wire. If so, it converts the logic/reg variable to a wire. VTW is enabled inside digital module and disabled at boundary with AMS and DMS connection.

For mixed signal designs, if discipline resolution or coercion happens in a particular connection, VTW optimization is turned off for that particular connection. Logic/Reg assigned in a procedural block (initial/always) is not converted to the wire.

The following example illustrates that with the `-dms_perf` option specified, `top.VOUT_VEC` is converted to wire.

```
rnm_coerce off". xrun top.sv -rnm_coerce off -dms_perf
-----
//top.sv
module top ();
  reg VOUT_VEC[0:1];
  ldo_monitor x1 (VOUT_VEC);
endmodule

module ldo_monitor (clk);
  input wire clk[0:1];
endmodule
```

The following example illustrates that though `-dms_perf` is specified, `top.VOUT_VEC` is not converted to wire because it is connected to `wrealsum`.

```
xrun top.sv -rnm_tech -dms_perf
-----
import cds_rnm_pkg::*;
module top ();
  reg VOUT_VEC;

  ldo_monitor x1 (VOUT_VEC);
endmodule

module ldo_monitor (clk);
  input wrealsum clk;
endmodule
```

Features Enabled Using the -adv_ms Option

You can now enable the following advanced mixed-signal verification functionalities using a single `xrun` command-line option, `-adv_ms`:

- [Connecting SystemVerilog Logic Ports to Electrical Using Wire/Interconnects](#)
- [Connecting SystemVerilog Variable Logic Vector to SPICE Bus](#)
- [Applying Bit-Wise Discipline for SPICE Bus Connections](#)

The `-adv_ms` option is a sub-set of the `-sv_ms` option; all features documented in mentioned in this topic (under `-adv_ms`) can be enabled using the `-sv_ms` option, as well. See also, [Features Enabled Using the -sv_ms Option](#).

Connecting SystemVerilog Logic Ports to Electrical Using Wire/Interconnects

In mixed-signal designs, to connect a SystemVerilog (SV) logic port to an Electrical port through a wire or interconnect, use the `-adv_ms` option.. The wire port connection can be any of the following types of SV logic port connections:

- Full bus to VAMS electrical using wire
- Partial bus/part-select/index part select to VAMS electrical using wire
- Bit-select to VAMS electrical using wire
- Concat to VAMS electrical using wire

The following example illustrates a SV logic vector connecting to an electrical port through wire:

```
//top.sv
module top;
  logic [1:0] bus;
  foo f1 (bus);
endmodule

// foo.vams
module foo(e);
  output [1:0] e;
```

```
child c0 (e[0]);
child c1 (e[1]);
endmodule

module child (p);
  output p;
  electrical p;
  real r;
analog begin
V(p) <+ r;
end

initial begin
  r = 0;
  # 10;
  r = 2.2;
end
endmodule
```

Related Topic

- [Features Enabled Using the -adv_ms Option](#)

Connecting SystemVerilog Variable Logic Vector to SPICE Bus

In mixed-signal designs, to connect a SystemVerilog logic vector to a SPICE electrical bus, use the `-adv_ms` option. The tool then generates a wire bus port as a wrapper and maps the wire bus to the electrical nodes.

The following is an example of an SV logic vector connecting to a SPICE electrical bus:

```
-----top.sv-----

`timescale 1ns/1ps

module top;
  real i[3:0];
  real o[3:0];
  spice_middle x0 ( i,o);
  //initial #3 $display("==== ", i,,o);

Endmodule

----- spice.scs-----

subckt spice_middle ( i_3 i_2 i_1 i_0 out\<3\> out\<2\> out\<1\> out\<0\>)
```

```
v7 ( i_3 0) vsource dc=5
v6 ( i_2 0) vsource dc=5
v5 ( i_1 0) vsource dc=5
v4 ( i_0 0) vsource dc=5
v3 (out\<3\> 0) vsource dc=5
v2 (out\<2\> 0) vsource dc=5
v1 (out\<1\> 0) vsource dc=5
v0 (out\<0\> 0) vsource dc=5
ends
```

Related Topics

- [Connecting SystemVerilog Logic Ports to Electrical Using Wire/Interconnects](#)
- [Features Enabled Using the -adv_ms Option](#)

Applying Bit-Wise Discipline for SPICE Bus Connections

In a SPICE port to bus element connection, when you use the `cellport` scopetype for cell port definitions, to apply an interface element parameter specification to specific connect modules, specify the `-adv_ms` option. This ensures that the parameter specification is applied only to the targeted connect module and not to every connect module on the bus.

The following example illustrates that discipline 1.8 is set to `top.in[0]`. And, the default discipline (`vsup=5`) is applied to the other ports `top.in[1]` and `top.in[2]`.

```
//top.sv
module top;
reg clk = 0;
wire ain;
wire [0:2] in;
always #5 clk=~clk;
assign in[0] = clk;
assign in[1] = clk;
assign in[2] = clk;
assign ain = clk;
spice_dut I0(in,ain); //SPICE Instance
endmodule
```

```
//ana_gate.sp
.subckt spice_dut in_0 in_1 in_2 ain
xinv0 in_0 out_0 inv_1p8
xinv1 in_1 out_1 inv_5
```

```
xinv2 in_2 out_2 inv_5
xinv3 ain aout inv_5
.ends

//amscf.scs
amsd {
portmap subckt=spice_dut autobus=yes file="./spice_dut.pb"
config cell=spice_dut use=spice
// IE Cards
}

amsd {
ie vsup=5.0
ie vsup=1.8 cellupport=spice_dut.in_0;
}
```

In the first case, discipline, 1.8 is set for top.in along with top.in[0]. In the second instance also, for top.in, top.in[0], and top.in[1] the discipline set is 1.8. The following ie card specification illustrates that when there is no default discipline set and only one bit is set, or multiple bits are set the same discipline, then the bus discipline is merged into one.

```
amsd {
    ie vsup=1.8 cellupport=spice_dut.in_0;
}
```

or

```
amsd {
    ie vsup=1.8 cellupport=spice_dut.in_0;
    ie vsup=1.8 cellupport=spice_dut.in_1;
}
```

The following ie card specification illustrates that when there is no default discipline and multiple bits are set to the different discipline, then no discipline is set to the unset bits. In this case, discipline 1.8 is set for top.in[0] and 2.8 is set for top.in[1].

```
amsd {
ie
vsup=1.8 cellupport=spice_dut.in_0;
ie
vsup=2.8 cellupport=spice_dut.in_1;
}
```

Related Topic

- [Features Enabled Using the -adv_ms Option](#)

Features Enabled Using the -sv_ms Option

You can now enable the following SystemVerilog mixed-signal verification functionalities using a single `xrun` command-line option, `-sv_ms`:

- [Connecting SystemVerilog Hierarchical UDT to Electrical](#)
- [Coercion for SystemVerilog Interfaces, Program, or Checker Blocks](#)
- [Using Wildcard-Named Port Connections in SV Bind Statements](#)
- [Connecting SystemVerilog Interface to DMS/Verilog-AMS \(Electrical\) Signals](#)
- [Connecting SystemVerilog Multi-Dimensional Array to SPICE/Electrical Ports](#)

The `-sv_ms` option is a super-set of the `-adv_ms` option; all features documented in the [Features Enabled Using the -adv_ms Option](#) topic can be enabled using the `-sv_ms` option also.

Connecting SystemVerilog Hierarchical UDT to Electrical

In addition to connecting SystemVerilog wreals to electrical nets, you can connect hierarchical user-defined types (UDTs) to Electrical nets. To connect hierarchical UDTs to electrical nets, use the `-sv_ms` option on the `xrun` command-line option. Hierarchical UDT including arrays of packed structure nettype, logic bit types (logic single type and bit select of logic bus), and L2E/E2L connect modules are supported. The following are the integer datatypes that are supported:

<code>shortint</code>	2-state data type, 16-bit signed integer
<code>int</code>	2-state data type, 32-bit signed integer
<code>longint</code>	2-state data type, 64-bit signed integer
<code>byte</code>	2-state data type, 8-bit signed integer or ASCII character
<code>bit</code>	2-state data type, user-defined vector size, unsigned
<code>logic</code>	4-state data type, user-defined vector size, unsigned

reg	4-state data type, user-defined vector size, unsigned
integer	4-state data type, 32-bit signed integer
time	4-state data type, 64-bit signed integer

L2E and E2L connect modules are used to setup hierarchical UDTs to Electrical port connections. However, LE_Bdir connect module is not supported because logic and reg types cannot connect to inout electrical/spice port.

Also, note that the following types of port connections are not supported:

- Connections containing multi-dimensional array
- Concatenation and replicate concat connections
- L/R related IE, or R/E related IE
- Connections to wreal/real ports, which lead to L/R interface elements

Consider the following example of a Verilog testbench where a hierarchical structure is defined. It also illustrates a UDT, `new_s` of the hierarchical structure connecting to two instances of electrical port, `inv`.

```
//top.sv

module top();
typedef struct packed {
    logic [1:0] a;
    logic [2:0] b;
} new_s;

new_s ns1;
typedef struct packed {
    logic aa;
    new_s new_s_var;
} new_s_level2;

new_s_level2 ns12;
logic [0:1] as1;
logic w_in;
logic w_in_2;
inv D_not2b (w_out, ns12.new_s_var.b[0]); // instance with
inv D_not2b_2 (w_out_2, ns12.new_s_var.a[1]);
...

endmodule
```


And, the following is an example of the AMSD control file containing SPICE information:

```
amsd {  
    // IE DEFINITIONS DEFAULT  
    ie vsup=1.6  
    // PARTITIONING  
    config cell=inv use=spice  
}
```

Related Topic

- [Features Enabled Using the -sv_ms Option](#)

Coercion for SystemVerilog Interfaces, Program, or Checker Blocks

The elaborator allows wreal coercion within SV interfaces and program blocks through the use of the `-rnm_tech` option or `ie card`. However, when an SV interface is instantiated within any of the following:

- another SystemVerilog interface (nested interfaces)
- a SystemVerilog program block
- a SystemVerilog checker

This wreal coercion (involving wreal datatypes) can result in real-to-logic connections inside the SV interface or program block, causing the simulator to output the following error message:

```
*E,CUINMI : connect module instantiation is not allowed inside an interface.
```

This error occurs because the elaborator is attempting to insert a connect module within the SV interface or program block, which is not allowed as per the IEEE 1800-2012 LRM. The IEEE 1800-2012 states specifically an SV interface or program block cannot instantiate modules and this restriction extends to connect modules. This restriction affects large complex designs where wreal coercion can be difficult to predict.

Therefore, you must use the `-sv_ms` option to enable wreal coercion in connections where SV interface is instantiated within another SV interface or SV checker is instantiated within an SV interface, or program block.

This improves how the elaborator executes wreal coercion within an SV interface or program block to prevent connect module insertion within them.

The following use case scenarios with examples describes how to use coercion within SV interfaces and program blocks. The examples mentioned use SV interfaces to illustrate the enhanced wreal coercion functionality.

- If a net is declared internally in an SV interface, program block, or checker or as an SV interface, program block, or checker port, it is coercible if all SV interface, program block, or checker instance port lists that it is directly connected to are also coercible. See [Use Case 1](#).
- If a nested SV interface, program block, or checker net/port is incoercible, a connect module is inserted at the boundary of the top most SV interface, program block, or checker to make the connection. However, connect modules are not inserted inside of an SV interface, program, or checker. See [Use Case 2](#).
- If an SV interface, program block, or checker net/port is incoercible and is directly connected to an SV interface, program block, or checker instance port that has a type mismatch, then normal SV port semantic rules are applied to connect the ports. And, no connect module is inserted. See [Use Case 4](#).

Use Case 1: Nested Interface Declaration with Coercible Wire Ports

In this example `top.IF2.y` and `top.IF2.IF1.x` are both coercible so both of them coerce to `wreal` because `top.z` coerces to `wreal` due to its connection to the real variable `top.C.z`. No connectmodule (CM) is required in this case as the whole wire path coerces to `wreal`.

```
interface inf1 (inout wire x);
endinterface

interface inf2 (inout wire y);
    inf1 IF1(y);
endinterface

module top();
    wire z;
    inf2 IF2(z);
    child C(z);
endmodule

module child (output var real r);
endmodule
```

Use Case 2: Nested Interface Declaration with Incoercible Wire Ports

In the following example `top.IF2.IF1.x` is incoercible because it is used in the sensitivity list of an always block. This means that `top.IF2.y` is incoercible because it is connected to `top.IF2.IF1.x`. The net `top.z` coerces to `wreal` due to its connection to the real port `top.C.r`. An R2L connect module is inserted in top to make the connection between `top.z` and `top.IF2.y`.

```
interface inf1 (inout wire x);
    always @(x) $display("X=%d", x);
endinterface

interface inf2 (inout wire y);
    inf1 IF1(y);
endinterface

module top();
    wire z;
    inf2 IF2(z);
    child C(z);
endmodule

module child (output var real r);
endmodule
```

Use Case 3: Nested Interface Declaration with Coercible Net and Real Var Connection

In the following example `top.IF2.y` coerces to `wreal` because all of its internal interface port connections are coercible. This means that `top.z` coerces to `wreal`. Since the entire path is real, no connect module is required.

```
interface inf1 (input real rr);
endinterface

interface inf2 (inout wire y);
    inf1 IF1(y);
endinterface

module top();
    wire z;
    inf2 IF2(z);
    child C(z);
endmodule
```

Use Case 4: Nested Interface Declaration with Incoercible Net and Real Var Connection

In the following example `top.IF2.y` is incoercible because it is used in the sensitivity list of an always block. No connect module is inserted between `top.IF2.y` and `top.IF2.IF1.x`. The SV port semantic rules are applied and the integral value of `top.IF2.y` is converted to real and assigned to `top.X`.

```
interface inf1 (input var real rr);
endinterface

interface inf2 (inout wire y);
    always @(y) $display("y=%d", y);
    inf1 IF1(y);
endinterface

module top();
    wire z;
    inf2 IF2(z);

    child C(z);
endmodule

module child (output var real r);
endmodule
```

Use Case 5: Non-Nested Coercible Direct Interface Port Connection in Module Scope

In the following example `top.IF1.x` and `top.IF2.y` should both coerce to `wreal` because both of them are coercible and `top.z` is connected to the real variable `top.C.r`. The entire path is real and no connect module is required.

```
interface inf1 (inout wire x);
endinterface

interface inf2 (inout wire y);
endinterface

module top();
    wire z;
    inf1 IF1(z);
    inf2 IF2(IF1.x);
    child C(z);
endmodule

module child (output var real r);
endmodule
```

Use Case 6: Non-Nested Incoercible Direct Interface Port Connection in Module Scope

In the following example `top.IF1.x` coerces to `wreal`; however, `top.IF2.y` is incoercible because it is used in the sensitivity list of an `always` block. A `connectmodule` is inserted in the scope of instance `top` between the OOMR `IF1.x` and `top.IF2.y`.

```
interface inf1 (inout wire x);
endinterface

interface inf2 (inout wire y);
    always @(y) $display("y=%d", y);
endinterface

module top();
    wire z;
    inf1 IF1(z)
    inf2 IF2(IF1.x);

    child C(z);
endmodule

module child (output var real r);
endmodule
```

Related Topic

- [Features Enabled Using the -sv_ms Option](#)

Using Wildcard-Named Port Connections in SV Bind Statements

Wildcard-named port connections represented using the dot-star notation (`.*`), is supported in the SV bind statements connecting to Verilog-AMS port connections. To enable the dot-star notation functionality, use the `xrun` command-line option, `-sv_ms`.

The following is an example of using the dot-star notation in a Verilog-AMS port connection:

```
% xrun -sv_ms amscf.scs vtop.vams top.sv inc.sv

//top.sv
module top();
vtop x1(clkin); <-- VAMS
bind vtop inc xinc(.*) <--- Target is VAMS
```

```
endmodule

//amscf.scs
amsd {
    ie vsup=1.8
}
//vtop.vams
module vtop();
electrical a;
wreal b;
...
endmodule

//inc.sv
module inc(a, b);
inout a;
inout b;
...
endmodule
```

Related Topic

- [Features Enabled Using the -sv_ms Option](#)

Connecting SystemVerilog Interface to DMS/Verilog-AMS (Electrical) Signals

A SystemVerilog (SV) interface may be used to connect modules with mixed-signal content. You must use the `-sv_ms` command-line option for the SV interface to electrical connections. For instance, for interface signals connecting to VAMS signals such as wreals or for complex interfaces where an interface signal is connecting to DMS reals.

The following are some of the limitations of using SV interface:

- There is limited support for coercion when using SystemVerilog interface in mixed-signal designs.
- Discipline resolution rules apply as usual, however, not all DR optimizations are supported.
- IE optimization, such as back-to-back IE insertion with electrical drivers and receivers is not supported. It is supported for back-to-back scenarios consisting of real type (`cds_rnm_pkg` types) drivers and real (`cds_rnm_pkg` type) receivers.

- Modports cannot be used with electrical signals.
- Ports involving interface signals that are unpacked array wires or user-defined nettypes are not supported.
- In designs with complex hierarchy of modules, the `-ams optie` option is not supported.
- Interface signals that are used in purely digital context remain digital even when connected to electrical signals. This includes the use of signal in clocking blocks.
- All limitations for digital implementations for SystemVerilog interface apply to mixed-signal implementations.

The following examples of interface declaration show how a mixed signal net can be declared as a member of an interface.

```
interface foo;
    user_nettype x;
    modport ...;
endinterface

interface foo;
    wrealldriver x;
endinterface

interface foo;
    wrealsum x;
endinterface

interface foo;
    wrealavg x;
endinterface

interface foo;
    wrealmax x;
endinterface

interface foo;
    wrealmin x;
endinterface

interface foo;
    wreal4state x;
endinterface
```

The following examples highlight how a SystemVerilog interface might be used to connect modules with mixed-signal SystemVerilog DMS objects (including custom nettype and modports)

Use Case 1: When the mixed-signal module instance has an interface port

```
//Interface Definition(SV)

interface foo;

    wreal4state x;

    wire x;

endinterface

//bar has an interface port
module bar(foo u);
endmodule

//baz has an interfafce port
module baz(foo u);
endmodule

module tb
    foo u()

//explicit port connection on instance v of bar
    bar v(u)
//implicit (or .*) connection on instance w of baz
    baz w(.*)
endmodule
```

Use Case 2: When the module instance does not have an interface port

```
//bak does not have an interface port

module bak (inout wreal4state x,
            inout wreal4state y);
endmodule

module tb;
foo u();
//explicit port connection on instance v of bar

    bar v(u);

//referenced port connection on instance w of bak
    bak w(u.x, u.y);
endmodule
```

The above cases outline the interface connectivity schemes when the modules have real numbered ports. These include wreals and SystemVerilog user-defined nettypes. Such modules are typically written in the SystemVerilog context. When the module is written in the Verilog-AMS context (file

with `.vams` extension), a SystemVerilog wrapper must be used to encapsulate the module before connection with an interface. This is because there is no concept of an interface in the Verilog-AMS language.

Use Case 3: When a module instance is a Verilog-AMS block

```
`include "disciplines.h"

//bal has an electrical port x and a wreal port y; it is a Verilog-AMS block
module bal(x,y);

    electrical x;
    wreal y;
end module

//balwrap is a SystemVerilog wrapper for bal
module balwrap(foo,u);
    bal v(u.x, u.y);
endmodule

module tb;

// implicit (or .*) connection on instance v or bar
    bar v(.*);
//implicit (or .*) connection on instance w of balwrap
    balwrap w(.*);
endmodule
```

Related Topic

- [Features Enabled Using the `-sv\_ms` Option](#)

Connecting SystemVerilog Multi-Dimensional Array to SPICE/Electrical Ports

To connect SystemVerilog multi-dimensional arrays to SPICE/Electrical ports, use the `-sv_ms` option. The expressions that connect to SPICE/Electrical at the parent-level/HDL scope can be bit-select, part-select, or a multi-dimensional array bus.

Currently, only the following two-dimensional arrays are supported:

- Packed by unpacked. For example, `wire [3:0] clk_out[0:2];`
- Packed. For example, `wire [3:0][0:2] clk_out;`

- Unpacked. For example, `wire clk_out[3:0][0:2];`

The following scenarios are not currently supported:

- Multi-dimensional arrays to `wreal` connection
- Multi-dimensional arrays of type `reg` to an electrical connection
- OOMR multi-dimensional array or a multi-dimensional array in an interface
- Array slice of multi-dimensional arrays to an electrical connection
- Multi-dimensional arrays when used in wildcard connections

Examples

The following example illustrates a bit-selected multi-dimensional array connection. The IE insertion happens at the array, `top.clk_out[5][12]` and SPICE port, `top.x1.p_clk`.

```
//top.sv
module testbench ();
wire [13:10] clk_out[2:7];
pll_top x1(.p_clk(clk_out[5][12]) );
endmodule

//analog.scs
.subckt pll_top p_clk
.ends
```

The following example illustrates a part-selected multi-dimensional array connection. The IE insertion happens between `top.clk_out[5][13]` and `top.x1.p0_clk_3`, `top.clk_out[5][12]` and `top.x1.p0_clk_2`, `top.clk_out[5][11]` and `top.x1.p0_clk_1`, and `top.clk_out[5][10]` and `top.x1.p0_clk_0`.

```
//top.sv
module testbench ();
wire [14:10] clk_out[2:7];
pll_top x1(.p0_clk({ p0_clk_3, p0_clk_2, p0_clk_1, p0_clk_0} ));
endmodule

//analog.scs
.subckt pll_top p0_clk_3 p0_clk_2 p0_clk_1 p0_clk_0
.ends

//generated skeleton:
module pll_top (.p0_clk( { p0_clk_3, p0_clk_2, p0_clk_1, p0_clk_0} ));
...
electrical p0_clk_3;
```

```
electrical p0_clk_2;  
electrical p0_clk_1;  
electrical p0_clk_0;  
endmodule
```

The following example illustrates a multi-dimensional array bus connection. The IE insertion happens between `top.clk_out[5][13]` and `top.x1.p0_clk_3`; `top.clk_out[5][12]` and `top.x1.p0_clk_2`; `top.clk_out[5][11]` and `top.x1.p0_clk_1`, and `top.clk_out[5][10]` and `s.`

```
//top.sv  
module testbench ();  
wire [13:10] clk_out[2:7];  
pll_top x1(.p0_clk(clk_out[5]));  
endmodule  
  
//analog.scs  
.subckt pll_top p0_clk_3 p0_clk_2 p0_clk_1 p0_clk_0  
.ends  
  
//generated skeleton:  
module pll_top (.p0_clk( { p0_clk_3, p0_clk_2, p0_clk_1,p0_clk_0} ));  
...  
electrical p0_clk_3;  
electrical p0_clk_2;  
electrical p0_clk_1;  
electrical p0_clk_0;  
endmodule
```

Related Topic

- [Features Enabled Using the -sv_ms Option](#)

Build Time & Run Time Performance Improvement for Mixed Signal Designs

In mixed-signal designs, you can use the `-dms_perf` command-line option to enhance the simulation and elaboration performance in the following use cases:

- When discipline resolution is enabled
- While specifying the scope-based coercion as off.
- Port connection check, especially when there is concatenation on parent or child

- When there is no SPICE OOMR in a design

To determine the optimization options that have been enabled using the `-dms_perf` option, see the generated `xrun` log file.

Related Topics

- [-rnm_coerce](#)
- [Variable To Wire \(VTW\) Optimization in Mixed-Signal Designs](#)

Low Power and Mixed-Signal Verification

With system-on-chip (SoC) design becoming increasingly more complex, and as process geometries continue to shrink, advanced power reduction strategies such as power shutoff (PSO), multi-voltage threshold (MVT) transistors, and dynamic voltage frequency scaling (DVFS) are now often required for low-power verification.

To support advanced power reduction in EDA tools, two power intent languages were developed. The Unified Power Standard (UPF) was approved by IEEE as the IEEE 1801 Standard and the Common Power Format (CPF) was approved by the Si2 coalition. These languages enable you to define the power intent for a design using semantics and constructs in a separate power intent file (UPF or CPF) independent of the hardware description language (HDL). All power-related specifications, constraints, and functional requirements are specified in this one file and these are then considered during the simulation, verification, and implementation of a design.

- You can specify the power intent using either the IEEE 1801 standard or CPF, but not both.
 - To enable the IEEE 1801 flow, you must add `-lps_1801 <file>.upf` on the `xrun` or `xmelab` command line.
 - To enable the CPF flow, you must add `-lps_cpf <file>.cpf` on the `xrun` or `xmelab` command line.

Related Topics

- [Using Common Power Format in Mixed-Signal Designs](#)
- [Using the IEEE 1801 Standard in Mixed-Signal Designs](#)

Using Common Power Format in Mixed-Signal Designs

If you have a digital design that uses Si2 Common Power Format (CPF) commands on some digital blocks, you can reuse the same CPF file when you reconfigure your design by replacing a digital block with an analog (SPICE) one. Using this feature, you can propagate the effect of power simulations onto your analog blocks in a direct way and measure analog effects, such as leakage current under power shutoff.

To enable the CPF flow, use `-lps_cpf <file>.cpf` on the `xrun` command line. You can use this feature for any mixed-signal design, with the restriction that CPF constructs cannot be applied directly to the analog content of your design. Under these conditions, the Xcelium simulator with mixed-signal option provides support for CPF and the following set of commands:

- `set_cpf_version`
- `set_design`
- `set_hierarchy_separator`
- `end_design`
- `set_instance`
- `set_macro_model`
- `end_macro_model`
- `create_isolation_rule`
- `create_ground_nets`
- `create_global_connection`
- `create_power_mode`
- `create_power_nets`
- `create_power_domain`
- `create_state_retention_rule`
- `create_nominal_condition`
- `create_mode_transition`
- `define_always_on_cell`

- `define_isolation_cell`
- `define_state_retention_cell`
- `update_power_domain`
- `create_level_shifter_rule`
- `update_isolation_rules`

CPF is a standard format, based on Tcl, for specifying all power-specific information and constraint requirements from design through verification and implementation.

Related Topics

- [Common Power Format User Guide](#)
- [Common Power Format Language Reference](#)
- [Low-Power Simulation Guide \(IEEE 1801\)](#)

Power-Smart Connect Modules for CPF

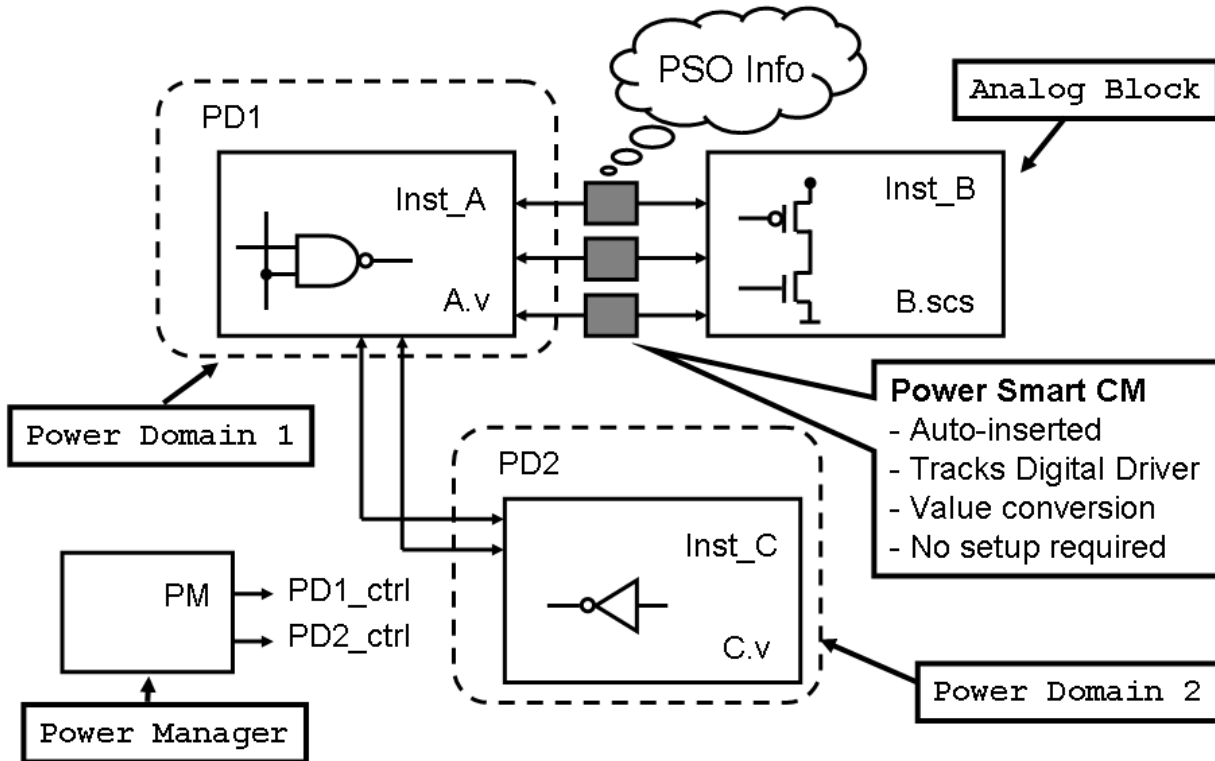
The Xcelium simulator with mixed-signal option supports CPF-influenced analog blocks by identifying them and inserting power-smart connect modules (CMs) during elaboration. The selection and insertion of power-smart connect modules complies with the Verilog-AMS language semantics. The connect rules in the Cadence installation have been enhanced to include power-smart connect modules. The power awareness of a connect module is determined by the tool on the basis of the connect module name (that is, the name ending with "LPS"). The power-smart connect modules (L2E/R2E) use an additional parameter named `vpso`. This parameter is used to specify the voltage value, which must be passed to the analog block when the digital block driving it is shut off. The default value of the `vpso` parameter is `0.2`.

You can also use the parameters `vtrlo`, `vtrhi`, and `vrtol` in power-smart connect modules to specify the ratio of low threshold, high threshold, and tolerance during value conversion.

Additionally, power-smart connect modules have been enhanced to support the current mode, in which electrical current is converted between wreal and electrical connection. When current mode processing is enabled for an LPS connect module, the behavior of the LPS connect modules is same as a regular connect module. And, the tool issues a warning that the current mode behavior is not low power-aware though the connect module is `*_LPS`.

Info: Current mode processing enabled for this low power aware connect module
(`tb.topInst.connect__E2R_LPS__ddiscrete_1_8`). Note that low power aspects are not expected to influence the current mode behavior

The following diagram illustrates how power-smart CMs carry the effect of digital power-shutoff (PSO) to analog blocks:



The design instances are **Inst_A**, **Inst_B**, **Inst_C**, and **PM**. **Inst_B** is an analog block connected to **Inst_A**. The dotted outlines indicate the two power domains, PD1 and PD2. **Inst_A** belongs to power domain 1 (PD1) and **Inst_C** belongs to power domain 2 (PD2). **PM** is a power manager module. The power control signals, **PD1_ctrl** and **PD2_ctrl**, correspond to power domains 1 and 2, respectively.

The power-smart L2R/E2R CMs carry the effect of PSO to RNM blocks with `wrealXState` and the power-smart E2L/R2L CMs carry the effect of PSO to digital blocks with `x`.

The Xcelium simulator with mixed-signal option allows the use of automatically inserted as well as manually inserted power-smart connect modules. The automatically inserted connect modules are selected by the tool based on the connect rules and connect libraries provided by the user. The manually inserted connect modules are explicitly instantiated by the user in the design and do not need any connect rules.

The Xcelium simulator with mixed-signal option supports designs containing VHDL-D drivers that directly or indirectly impact power-smart connect modules in the low-power mixed-signal simulation. However, this support is currently not available for VHDL-SPICE cases where VHDL conversion

elements (CEs) are used.

Also, the power-smart connect modules support advanced power-domain features like power modes and nominal conditions, resulting in improved digital-to-analog transition inside the modules in low power mixed-signal simulation. A power-smart connect module in AMS-CPF can:

- Identify and honor all possible methods of shutting off a power domain.
- Identify a change in the active state of a power domain as soon as the power mode transition occurs and read the voltage value from the associated nominal condition.
- Use the voltage value of a nominal condition associated with a power domain to perform the digital-to-analog transition.

Refer to the connect library in the installation to see the power-smart connect module definitions.

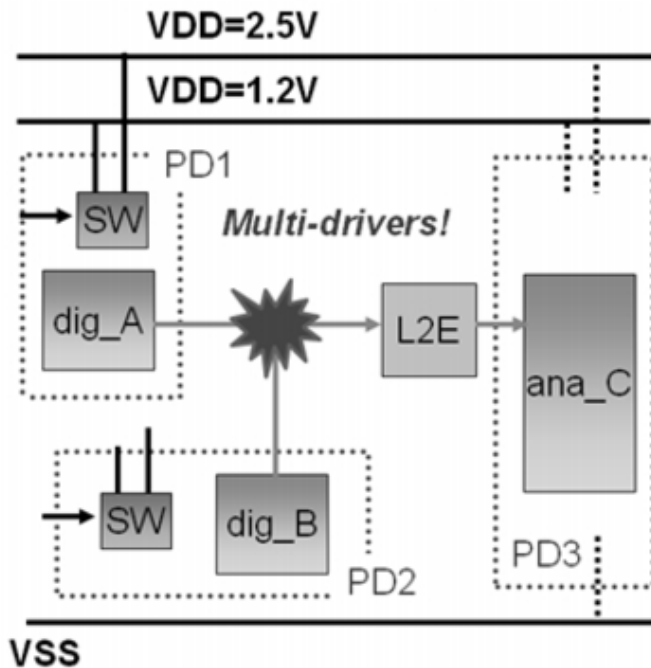
Related Topic

- [Power Smart Connect Modules for IEEE 1801](#)

Support for Multiple Digital Drivers

Consider a situation where an analog signal is driven by multiple digital drivers that take different nominal conditions from one or more power domains, such as the example shown below.

Modules `dig_A` and `dig_B` belong to power domains `PD1` and `PD2`, both of which are switchable power domains with a nominal condition of `1.2V` and `2.5V`. When the two modules drive the same port of an analog block `ana_C`, it is the case of multiple digital drivers in AMS-CPF.



When multiple digital drivers carrying CPF information drive a single analog port, the following describes how the multiple digital drivers are resolved:

- If all of these drivers belong to the same power domain: Continue the simulation without any warnings.
- If there is one or more power domains in power on or standby state: Report a warning, select the maximum nominal on or standby voltage, and apply it on the analog side.
- If there is one or more power domains in power off state: Report a warning, select the maximum nominal off voltage as the v_{ps0} , and apply it on the analog side.
- If there is one or more power domains in power transition state: Report a warning and apply a random voltage on the analog side.

Here is an example of a warning message:

```
AMSCPF_VPI WARNING (at simtime 900.000000ns) : Multiple digital drivers are not allowed
with a power smart connect module (top.connect__L2E_LPS__ddiscrete_2_4.Din), only one will
be used:
```

```
PowerDomain : PD_Dig1 - Voltage : 1.800000 (selected)
PowerDomain : PD_Dig2 - Voltage : 1.600000
```

Related Topic

- [Using Common Power Format in Mixed-Signal Designs](#)

ie Card Parameters for Power-Smart Connect Modules

The `ie` card supports parameters like X-voltage that are specific to the power-smart connect module. An additional parameter called `vpso` on the `ie` card can also be used to indicate the voltage value for power shut off voltage. You can also use the following parameters on the `ie` card:

- `vpso`: Specifies the voltage value for power shut off voltage. The `vpso` value overrides the `vx_amscpf_poweroff` parameter in the power-smart connect module.
- `vtrlo`: Specifies the ratio of low threshold during value conversion. The default value is `vtlo/vsup`.
- `vtrhi`: Specifies the ratio of high threshold during value conversion. The default value is `vthi/vsup`.
- `vrtol`: Specifies the ratio of tolerance during value conversion. The default value is `vtlo/vsup`.
- `currentmode`: Enables wreal current mode processing for low power simulation. The default value is 0. The tool issues a warning that the current mode behavior is not low power aware though the connect module is `_LPS`.

The threshold value and tolerance value are calculated based on the ratio and the supply voltage of power domain. You can specify either the ratio (`vtrhi=2/3 vtrlo=1/3 vrtol=1/12`) or the threshold directly (`vthi=1.2 vtlo=0.6 vtol=0.1`).

The following is an example of the parameters you can use in the `ie` card statement for power-smart connect modules:

```
amsd{  
  
  ie vsup=1.8 instport="testbench.vlog_buf.I1.in" vpso=0.1, vthi=1.2 vtlo=0.6 vtol=0.1  
  currentmode=1,  
  
}
```

Related Topic

- [amsd Block](#)

Power Aware Modeling in Common Power Format

A set of built-in Verilog system tasks and VHDL procedures can be used in Verilog (-AMS) or VHDL(-AMS) code to obtain information about the low-power simulation. Most of the tasks / procedures link a Verilog register or a VHDL signal to some low-power information, such as the power-down state of a power domain, the name of the power mode that a domain has just entered, the voltage that a power domain is at after a nominal condition transition, and so on.

Following are the Verilog system tasks and functions:

```
$lps_get_power_domain(<power_domain_register>[, <instance>]);  
$lps_link_power_domain_powerdown(<link_register> [, <power_domain>);  
$lps_link_power_domain_standby(<link_register>[, <power_domain>]);  
$lps_link_power_domain_voltage(<link_variable>[, <power_domain>]);  
$lps_link_power_domain_gnd_voltage(<link_variable> [, <power_domain>]);  
$lps_link_power_domain_nmos_voltage(<link_variable> [, <power_domain>]);  
$lps_link_power_domain_pmos_voltage(<link_variable> [, <power_domain>]);  
$lps_link_power_domain_nominal_condition(<link_register> [, <power_domain>]);  
$lps_link_power_domain_power_mode(<link_register> [, <power_domain>]);  
$lps_enabled();  
$lps_get_stime();
```

One limitation of low power system task is that you cannot use them inside an analog procedural block. For example, the following module definition will not work because the low-power system task `$lps_enabled()` is used inside the `analog` procedural block.

```
module top (out);  
    inout      out;  
    reg        temp = 0;  
    electrical out;  
    real p;  
  
    initial  
        begin  
            #10 temp = 1;  
        end  
  
    analog  
        begin  
            if (!$lps_enabled && (temp == 0))  
                p = 0.0;  
            else  
                p = 3.0;  
            V(out) <+ p;  
        end  
endmodule
```

Related Topic

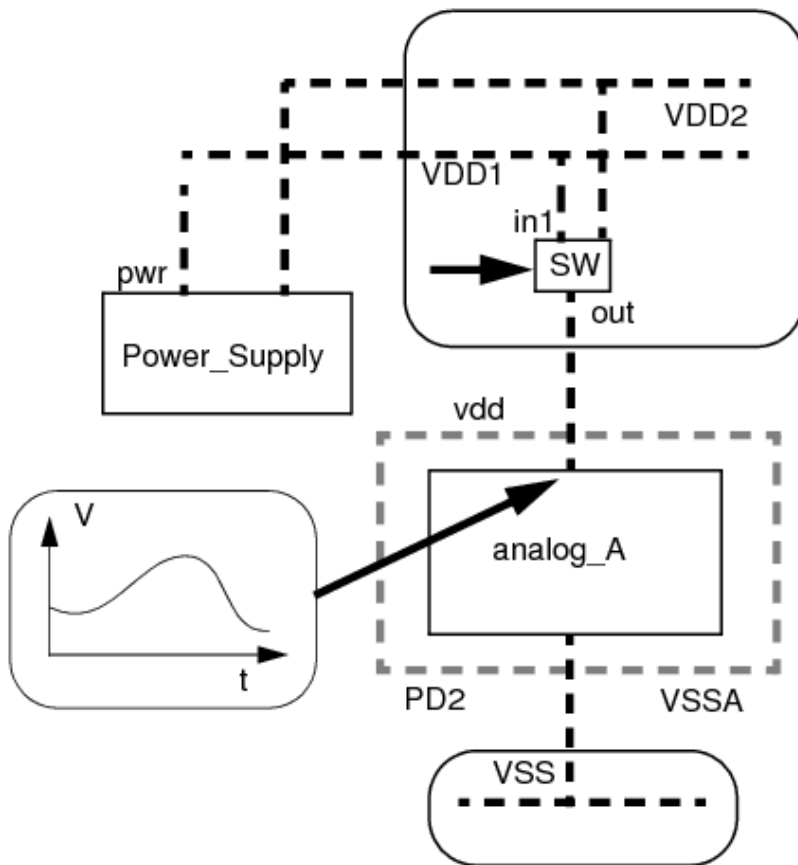
- [Power Aware Modeling](#)

Connecting Design Supply Net to CPF Power Domain and Design Net

You can define a connection on mixed-signal power-supply nets based on the CPF command `create_global_connection`. The power net or hierarchical net is defined either in the CPF scope by using the `create_power_nets` command, or it is defined in the HDL scope. And, the ground connections for the power domain net or hierarchical nets can be specified in the CPF scope using the `create_ground_nets` command. You can associate the power supply nets to the power domain by using the CPF command `update_power_domain`.

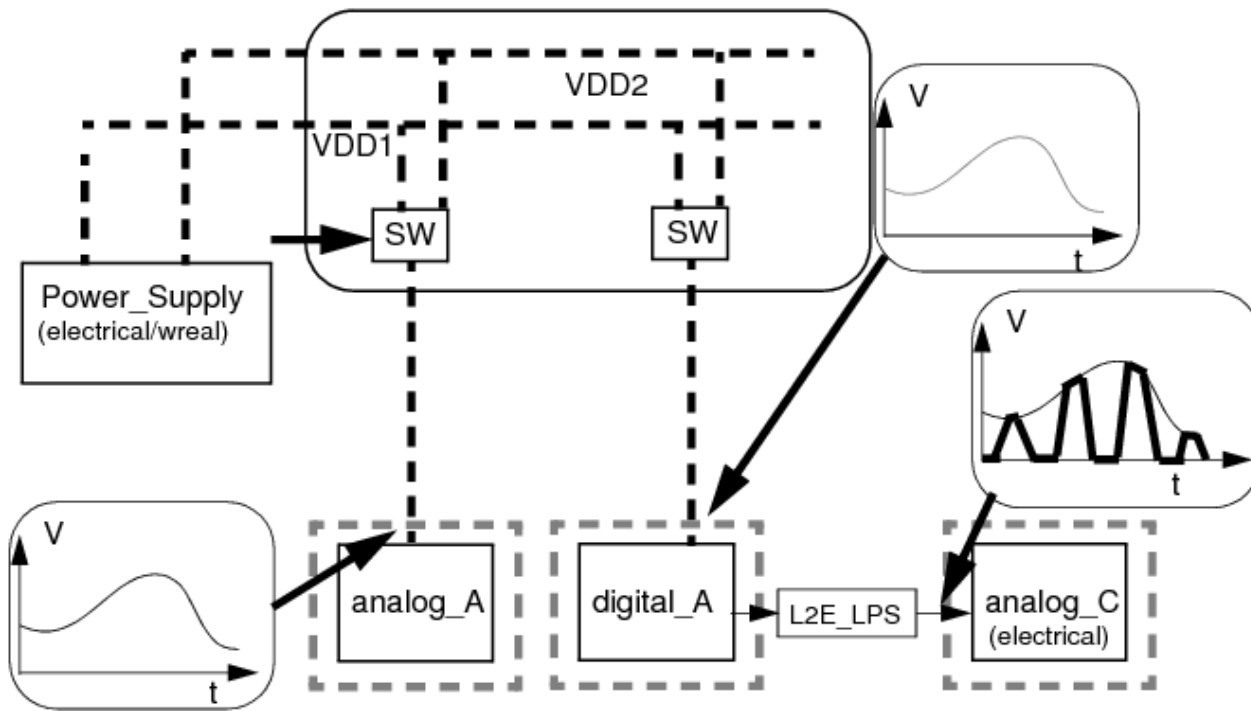
To enable this feature, use the `xrun` command-line option, `-lps_ams_connect_supply`.

The following figure shows the connection between a design supply net and the CPF power domain and design net:



```
create_power_nets -nets VDD1 VDDA
create_global_connection \
-net VDD1 \
-pins { Power_Supply/pwr SW/in1 }
create_global_connection \
-net VDDA \
-pins { SW/out analog_A/vdd }
update_power_domain \
-name PD2 \
-primary_power_net VDDA
```

In a digital instance, which is under a power domain driven by design supply net, the power aware connect module inserted on the instance port is aware of the design supply net value dynamically.



To set the value of net for domain mapped nets that are also physically connected at parent level, you must use the `-lps_ams_highconn_relax` option. This also enables power supply net connectivity on both HDL and CPF, if supply connection exists on both.

In the following example, the tool applies the power supply connection on the top level port of this signal.

```
create_global_connection -net VDD -pins VDD -instances {testbench/PMUInst digInst anaInst wrvInst}
create_global_connection -net VSS -pins VSS -instances {testbench/PMUInst digInst anaInst wrvInst}
=====test.vams=====
module top(VDD, VSS);
input VDD, VSS;
electrical VDD;
wreal VSS;
dinv digInst(VDD, VSS);
ainv anaInst(VDD, VSS);
winv wrvInst(VDD, VSS);
endmodule

module testbench;
top dut(vdd, vss);
PMU PMUInst();
endmodule
```

```
module dinv (VDD, VSS);
input VDD, VSS;
endmodule

module ainv (VDD, VSS);
input VDD, VSS;
electrical VDD, VSS;
endmodule

module winv (VDD, VSS);
input VDD, VSS;
wreal VDD, VSS;
endmodule
```

The net `testbench.PMUIInst.VDD` and `testbench.PMUIInst.VSS` are connected with the port `VDD` and `VSS` of instances `anaInst` and `wrvInst`, respectively, through CPF supply nets `VDD` and `VSS`. At the same time the port `VDD` and `VSS` of the instances `digInst`, `anaInst`, and `wrvInst` have physical high connection in HDL source code, which introduces the multiple drivers on this net.

You can also establish a connection on mixed-signal power-supply nets, including SV-UDN nets, by using the domain mapping between macro model instantiation.

For SV-UDN power-supply nets, the UDNs connecting to the same supply net must have the same nettype. Also, for the electrical to UDN connections, you must provide the connect module specifications through `ie card` or connect rules.

The tool automatically creates the internal power nets if the power/ground pins of the power domains are specified using the CPF command `update_power_domain`, as shown below.

```
update_power_domain -name VREG_OUT -primary_power_net {out} -primary_ground_net {gnd}
```

The power net connection can be established by using the `-domain_mapping` command, as shown below.

```
set_instance top.vreg_inst25 -model VREG -domain_mapping { {VREG_OUT PD_25} }
```

Here is an example:

```
create_power_domain -name PD_25
set_macro_model VREG
    create_power_domain -name VREG_OUT
    update_power_domain -name VREG_OUT -primary_power_net {out}-primary_ground_net {gnd}
end_macro_model

set_macro_model DECODER
    create_power_domain -name PD_DECODER
    update_power_domain -name PD_DECODER -primary_power_net {vdd} -primary_ground_net
```



```
{vss}
end_macro_model

set_instance top.vreg_inst25 -model VREG -domain_mapping { {VREG_OUT PD_25} }
set_instance top.dec_inst25 -model DECODER -domain_mapping {{PD_DECODER PD_25}}
```

Based on the above CPF definition, supply connection between `top.vreg_inst25.out->top.dec_inst25.vdd` and `top.vreg_inst25.out->top.dec_inst25.vss` is applied automatically.

Related Topic

- [Using CPF Supply Connection in Mixed-Signal Simulation.](#)

Support for Transition Slope of Power Supply in CPF-Controlled Analog Block

For CPF-aware mixed-signal simulation, AMS automatically provides a dynamic power supply for analog (electrical or wreal models) blocks that are controlled by CPF specifications. The power supply value changes at the start point of the power transition. For electrical power supply, the simulator models a transition slope and transition latency during the transition state with the `rise_time` and `fall_time` of transition statement, and applies the maximum slope and time value. For wreal or real power supply, the voltage changes at the endpoint of the transition, which enables you to determine whether the wreal power supply is in the transition period.



If the power supply net is specified on a VHDL net, the net is removed from the auto supply netlist. In addition, the transition slope on real or wreal supply net is ignored.

Referring to the Power Supply of Smart IE from Analog Side Locally

In AMS or DMS simulation, the reference voltage of analog-to-digital (A-to-D) or digital-to-analog (D-to-A) conversion is always determined by working voltage of digital side. In case if a voltage conflict is detected in the mixed boundary, the Xcelium simulator with mixed-signal option automatically inserts a level shifter with the Interface Element (IE), based on the level shifter rules specified in CPF file (using the `create_level_shifter_rule` command) and the reference voltage of IE is automatically linked to nominal voltage of analog block.

This feature is enabled by specifying the `-lps_ams_lsr` command-line option. In addition, it is not applicable for power-smart R2E CMs.

Checking Conflicting Power Domains on Mixed-Signal Boundary

Multiple power domains connecting to the same connect module on the digital or analog side result in a voltage conflict. The Xcelium simulator with mixed-signal option checks for such voltage conflicts during the elaboration stage and generates an error if:

- A power-smart IE connects to the digital drivers with different power domains.

```
xmelab: *E,PIEMLDR: [LPS] Multiple digital drivers from different power domains are
connected to same mixed-signal net
```

- A power-smart IE connects to the digital loads with different power domains.

```
xmelab: *E,PIEMLLD: [LPS] Multiple digital loads from different power domains are
connected to same mixed-signal net
```

- A power-smart IE connects to the analog signals (drivers or loads) with different power domains.

```
xmelab: *E,PIEMASG: [LPS] Multiple analog signals from different power domains are
connected to same mixed-signal net
```

- A power-smart IE connects to wreal or real signals (drivers and loads) with different power domains.

```
xmelab: *E,PIEMRSG: [LPS] Multiple real valued signals from different power domains
are connected to same mixed-signal net
```

To downgrade these errors to a warning, you can use the `-xmwarn` option with the error message ID (`-xmwarn PIEMLLD`) and re-run the design.

The simulator does not generate an error if the digital driver and load belong to different power domains.

This feature works only when the power domain voltage is specified in the CPF file and enabled for low power simulation. You must specify the following:

- `-lps_pmode` **or** `-lps_mvs` **or** `-lps_ams_connect_supply`
- `-lps_ams_sim` **or** the boundary port domain specified on the analog or wreal port

In addition, you must specify the following in the CPF file:

- If `-lps_pmode` **or** `-lps_mvs` is enabled, you must specify:
 - `create_nominal_condition`
 - `create_power_domain`, `-active_state_condition`, **or** `create_power_mode`
- If `-lps_ams_connect_supply` is enabled, you must specify `update_power_domain`, `-primary_power_net`, **or** `-primary_ground_net`

Using wreal Data Type in Mixed-Signal Designs with CPF

The `wreal` data type can be used in AMS design with CPF. When a wreal module is power-shut-off, the wreal signal in this module is forced to `wrealXState`.

To change this behavior, use the following two options:

- `-lps_wreal_nocorrupt`: Prevents the forcing of the wreal signal and also prevents the original value from getting corrupted.
- `-lps_wreal_corrupt_value <value>`: Keeps the behavior consistent with the behavior of the analog signal from power smart [IE](#). The user can pick a value of choice, and also restore the backward compatible behavior of 0.0.

You can also apply the power corruption value locally on wreal signals during the power-off state by adding the following options in the `user_attributes` argument of the `update_power_domain` CPF command:

- `wreal_nocorrupt [ instance_name | wire_name ]`: Specifies that the objects of a power domain are non-corrupt during the simulation and maintains their values.
- `wreal_corrupt_value [ value | value@object ]`: Specifies the corruption value of objects in the power domain.

Power Corruption on the Boundary Port of a Wreal Model

Power corruption is applied on the wreal boundary port of a macro model specified in the CPF file, for both input and output ports. For example:

```
set_macro_model wr1
create_power_domain -name pd1 -boundary_ports {rout1} -default
create_power_domain -name pd2 -boundary_ports {rout2}
end_macro_model
```

This feature is controlled by the option `-lps_wreal_bport_corruption`. If the `-lps_wreal_bport_corruption` option is specified, the wreal boundary ports that are specified in the CPF file and are associated with the power-shutoff domains are forced to the power corruption value.

The power corruption value in the power-shutoff state is determined by the `-lps_wreal_nocorrupt` or `-lps_wreal_corrupt_value` options, or the `wreal_nocorrupt` or `wreal_corrupt_value` options specified using the `user_attributes` argument of the `update_power_domain` CPF command, as shown below.

```
update_power_domain -name pd1 -user_attributes { wreal_corrupt_value 0.2 }
update_power_domain -name pd2 -user_attributes { wreal_corrupt_value {0.5@inst1} \
                                                wreal_corrupt_value
                                                {wrealZState@inst2.rout} \
                                                wreal_nocorrupt {inst3} }
```

Wreal instances that are specified using the `create_power_domain` CPF command also have the power corruption value applied to them.

Wreal Expressions in CPF

Wreal expressions are supported in the following CPF commands that require a control condition:

- `create_power_domain`
- `create_mode_transition`
- `create_isolation_rule`
- `create_state_retention_rule`

The signal in the wreal expression must be real valued, otherwise, the simulator generates an error. Currently, only the wreal signal and the wire which is coerced as wreal in Verilog are supported. To enable the support for wreal expressions in CPF you must specify the `-lps_ams_sim` option.

The following operators are supported in a wreal expression:

- `<`, `<=`, `>`, `>=`
- `==` and `!=`
- `===` and `!==`

The case equality operators `===` and `!==` can only be followed by `wrealXState` and `wrealZState`. In addition, `wrealXState` and `wrealZState` can only be specified with the case equality operators. For example:

```
create_power_domain -name PD -shutoff_condition pmc.vsup===wrealXState
```

The following conditions apply to wreal expressions:

- The reference value should be a real number, `wrealXState` or `wrealZState`. Variables or parameters are not supported.
- The wreal expression returns the value zero if the specified relation is false or one if the specified relation is true.
- The wreal expression can be combined with a boolean expression. For example: `pmc.pso>=0.2 !(pmc.pso>=0.2) pmc.pso&&(pmc.pso>=0.2)`
- `wrealXState` or `wrealZState` on real valued signal is supported, and the expression is resolved as logic `x`. For example, in the example below, if `pmc.pso` is `wrealXState` or `wrealZState`, the shutoff condition signal is in `x` state.

```
create_power_domain -name PD -shutoff_condition (pmc.pso>=0.2)
```

- The wreal expression can be specified in a virtual port using the `-port_mapping` argument of the `set_instance` command, as shown below.

```
set_instance digInst -port_mapping {{virpso pmc.vsup<0.2}}
set_design dig_child -ports {virpso}
create_power_domain -name PD0
create_power_domain -name PD1
-shutoff_condition virpso
```

```
end_design
```

Related Topic

- [Specifying Boundary Ports and Macro Models](#)

Specifying Boundary Ports and Macro Models

Boundary ports are specified and applied to primary inputs or outputs of a top-level design or a macro model.

If a mixed-signal net is connected to a boundary port, AMS-CPF ensures that the:

- ISO device is inserted into the boundary port
- ISO device correctly connects with the digital port of the connect module

During the elaboration phase (`xrun -elaborate`), AMS-CPF:

- replaces the regular connect module with the power smart connect module if the digital side or the analog side is specified as a boundary port
- creates an isolation device for the AMS Boundary port if the driver and load belong to different digital and electrical scopes
- connects the ISO device with the digital side of the connect module during driver-receiver segregation

During the simulation phase (`xrun -r`), AMS-CPF checks the power status of the boundary port domain in the `amscpf-vpi` function.

Bus, bit-select, and part-select of an analog port, including the SPICE port, can be used in the boundary port while performing domain mapping, especially for macro model usage, as shown in the following example:

```
//SPICE block
.subckt ana_inv in[3] in[2] in[1] in[0] out[3] out[2] out[1] out[0] vdd vss
...
.ends ana_inv
//instantiation in top
module top();
...

wire [3:0]data1;
wire [3:0]data2;
```

```
wire [3:0]aout1;
wire [3:0]aout2;

ana_inv anaInst1 (data1, aout1, vdd, gnd);
...
endmodule

//Verilog ref block
module ana_inv (in,out,vdd,vss);
input [3:0]in;
output [3:0]out;
electrical [3:0]in;
electrical [3:0]out;
...
endmodule
```

The boundary port of a SPICE instance can be defined in the CPF file as follows:

```
# design ana_inv
set_macro_model_ams_inv
create_power_domain -name PD_ON -boundary_ports { vdd vss } -default
create_power_domain -name PD_SW -boundary_ports { in out[3] out[2:1] out[0] }
end_macro_model
```

Here,

- The whole bus port, such as `in`, can be specified.
- Part-select of a bus port is supported. For example, `out[2:1]`.

The index order of the part-select expression should be in the same direction as the bus definition. `out[1:2]` would be illegal in the above example.

- Bit-select of a bus port is supported. For example, `out[0]`.

Using Electrical Expressions in CPF

Electrical expressions are supported with the CPF commands that require power event control conditions. To enable the support for electrical expressions in CPF you must specify the –

`lps_ams_analog_ccnd` command-line option. The electrical signal could be in a VAMS or SPICE model and can be specified in power event control condition directly or in boolean expression of power event control condition.

The tool collects the electrical signals that are used in CPF control condition and creates internal wreal signals for each electrical one. Then, the tool relaunches elaborator to compile the internal wreal signals and automatically inserts one of the following connect modules:

- An `ER_bidir` connect module to convert the electrical voltage value to internal `wreal` nets. The tool replaces the electrical signals in CPF control condition with the internal `wreal` one and reuses the same processing of `wreal` expression as CPF control condition. A mapping between original electrical signal and internal `wreal` signal is generated.
- An `E2L_2_LPS` connect module, if there is an isolation power event control condition on the electrical signal. The electrical voltage value is converted to logic. The tool detects the power domain definition on the electrical signal of CPF control condition; and, uses the working voltage of this domain as reference. The working voltage can be from dynamic supply connection through domain mapping, CPF global connection, or static voltage info of nominal condition.

If there is no power domain definition detected on the electrical signal, the tool uses the reference voltage info defined in `ie` card. And, if there is no `ie` card specified, the default reference value, 1.8 is used.

The current mode and the electrical signal in IEEE1801 (UPF) control condition are not supported.

The following operators are supported in the expressions:

- `>`, `>=`, `<`, `<=`
- `==` and `!=`
- `==` and `!==`

The case equality operators `===` and `!==` can only be followed by ``wrealXState` and ``wrealZState`. For example:

```
create_power_domain -name PD -shutoff_condition pmc.vsup===`wrealXState
```

Also, the following conditions must apply:

The equality on real value means that the value is exact same for computer system. You must consider the tolerance in compare.

The reference value should be a real number or ``wrealXState`/``wrealZState`. Variable or parameters are not supported.

The ``wrealXState`/``wrealZState` must follow with case equality operators only.

Examples

In following example `VDD` and `AVDD [0]` electrical signals are used as shutoff conditions as per the CPF description. An `ER_bidir` connect module is inserted as shown below.


```
#####
### HDL description          ###
#####

module PMU();
    electrical VDD, VSS;
    electrical [1:0]aVDD, aVSS;
    electrical gnd;          ground gnd;

    vsource #(.dc(0.0), .type("dc")) vgn0 (VSS, gnd);
    analog begin
        V(VDD)  <+ transition(vsup, 0.0, 20e-9, 30e-9, 0.0);
        V(aVDD[0]) <+ transition(vsup1, 0.0, 20e-9, 30e-9, 0.0);
    end
endmodule

#####
### CPF description  ###
#####

create_power_domain -name PDD -default
create_power_domain -name PD1 -base_domains PDD \
    -instances {digInst1} \
    -shutoff_condition {testbench/PMUInst/VDD<=0.5}

create_power_domain -name PD2 -base_domains PDD \
    -instances {digInst2} \
    -shutoff_condition {testbench/PMUInst/aVDD[0]<=0.5}
```

In the following example, electrical signals `avdd`, `avdd1`, and `aiso` are defined in HDL to drive CPF power event. The power domain `PDD` is associated with electrical supply net `avdd` and `avss`. And, the electrical signal `avdd` is used to drive domain shutoff condition. The electrical signals `avdd` belongs to domain `PDD`. The voltage of `PDD` is used as reference of electrical signal `avdd`.

The power domain `PD1` is associated with electrical supply net `avdd1` and `avss`. And the electrical signal `avdd1` is used to drive domain shutoff condition. The electrical signals `avdd1` belongs to domain `PDD`. The voltage of `PDD` is used as reference of electrical signal `avdd1`.

Isolation rules `ISO1` and `ISO2` are control by electrical `aiso`. The electrical signals `aiso` belongs to domain `PD1`. The voltage of `PD1` is used as reference of electrical signal `aiso`.

```
#####
### HDL description          ###
#####

module pmu(avdd,avdd1,avss,aiso);
    output avdd, avdd1, avss, aiso;
    electrical avdd, avdd1, avss, aiso;
    ground avss;
```

```
analog begin
    ...
end
endmodule

#####
### CPF description ###
#####

set_macro_model pmu
    create_power_domain -name MPD0 -default
    update_power_domain -name MPD0 -primary_power_net avdd -primary_ground_net avss
    create_power_domain -name MPD1 -boundary_ports { aiso }
    update_power_domain -name MPD1 -primary_power_net avdd1 -primary_ground_net avss
end_macro_model pmu

set_design testbench -testbench

create_power_nets -nets { avdd avdd1 }
create_ground_nets -nets { avss }

    create_power_domain -name PDD -default \
        -shutoff_condition {!testbench/ipmu/avdd}

create_power_domain -name PD1 -base_domains PDD \
    -shutoff_condition {!testbench/ipmu/avdd1}

update_power_domain -name PDD -primary_power_net avdd -primary_ground_net avss
update_power_domain -name PD1 -primary_power_net avdd1 -primary_ground_net avss

set_instance dut -domain_mapping { {PDD PDD} {PD1 PD1} }
set_design top

create_power_domain -name PDD -default
create_power_domain -name PD1 -base_domains PDD \
    -instances {digInst}

create_isolation_rule -name ISO1 -to PD1 -isolation_output high \
    -isolation_condition {!testbench/ipmu/aiso}

create_isolation_rule -name ISO2 -from PD1 -isolation_output low \
    -isolation_condition {!testbench/ipmu/aiso}

end_design top

set_instance ipmu -model pmu -domain_mapping { {MPD0 PDD} {MPD1 PD1} }
end_design testbench
```

Feedthrough Wire Analysis

Technology improvements--related to tracking the drivers and loads on a net more accurately, thus enabling more accurate low-power semantics for a net--have been enabled in feedthrough wire analysis for AMS Designs. The following are the consequences of the improved behavior:

- ISO devices are not created for a floating net, that is, a net without driver and load.
- If logical expressions are embedded in a continuous assignment statement, both the driver and receiver of that assignment statement are corrupted when that block is in the power shut-off mode. This is different from the behavior in 11.1, where only the receiver was corrupted.

Example:

```
module dig_child (lin);  
  inout lin;  
  
  ...  
  
  assign ltmp = ~lin;  
endmodule
```

The continuous assignment statement `assign ltmp = ~lin;` contains a bitwise negation operator, based on which the nets `ltmp` and `lin` do not qualify as feedthrough wires. The nets are forced to x state when the `dig_child` block is in the power shut-off mode.

For more information about feedthrough analysis, refer to the Feedthrough Wires section in the *Low-Power Simulation Guide*.

Using the IEEE 1801 Standard in Mixed-Signal Designs

The Xcelium simulator supports the following versions of the IEEE 1801 standard in a mixed-signal context:

- UPF 2.0 = IEEE 1801-2009
- UPF 2.1 = IEEE 1801-2013
- UPF 3.0 = IEEE 1801-2015
- UPF 3.1 = IEEE 1801-2018

The Xcelium simulator with the mixed-signal option supports the Unified Power Format (UPF) commands listed in the following table. You must use the IEEE 1801 power intent to specify these commands. This implements power-saving techniques early on in the design process and can help to

deliver an end-to-end low-power design solution. For more detailed information about these commands, see the [Low-Power Simulation Guide \(IEEE 1801\)](#).

You can enable the IEEE 1801 flow using `-lps_1801 <file>.upf` on the `xrun` or `xmelab` command line.

UPF Commands	Description
<code>add_port_state</code>	Specifies a state for a UPF supply port. Once specified, this state can then be used in an <code>add_pst_state -state</code> expression to build a power state table.
<code>add_pst_state</code>	Specifies a power state for each supply net defined in the power state table (PST). The total number of supply net states must equal the total number of supply nets listed for the corresponding <code>create_pst</code> command. If the number of states and nets differs, then the simulator generates an error.
<code>add_power_state</code>	Explicitly specifies one or more power states for a supply set, power domain, composite domain, group, or instance. A power state can include either a supply expression, logic expression, and/or a simstate definition. By default, any state specified using this command is legal.
<code>associate_supply_set</code>	Specifies to associate two or more supply sets. By default, Xcelium supports associating supply sets using IEEE 1801-2015 syntax and semantics. This command does not change the power state defined for each supply set.
<code>connect_logic_net</code>	Connects a logic net to specified ports. The net is propagated through implicitly created ports and nets throughout the logic hierarchy in the descendant tree of the active scope as required.
<code>connect_supply_net</code>	Connects a supply net to one or more ports.
<code>connect_supply_set</code>	Connects a supply set to one or more design elements.
<code>create_logic_net</code>	Defines a logic net in the active scope, where the net name is a simple (non-hierarchical) name.
<code>create_logic_port</code>	Defines a logic port in the active scope, where the port name is a simple (non-hierarchical) port. Xcelium supports both implicit and explicit logic port connections.

<code>create_pst</code>	Defines a name for the power state table (PST) and specifies a list of one or more UPF supply nets. You can use the <code>add_pst_state</code> command to specify a state for each supply net in an 1801 design.
<code>create_power_domain</code>	Creates a power domain. In UPF 1.0, this command specifies only the instances that comprise the domain.
<code>create_power_switch</code>	Defines a power switch. This command defines an instance of a power switch in the active scope or in the scope of the power domain specified with the <code>-domain</code> option.
<code>create_supply_net</code>	Defines a supply net in the active scope or in the scope of the specified domain.
<code>create_supply_port</code>	Defines a supply port in the active scope or in the scope of a specified domain.
<code>create_supply_set</code>	Creates or updates a supply set in the active scope. A supply set with no explicit functions specified is supported by Xcelium as an implicit handle.
<code>find_objects</code>	Creates queries in the specified scope to search for and return 1801 design elements, nets, or ports that you want to use as arguments to other 1801 commands.
<code>load_upf</code>	Executes the commands in the specified 1801 file.
<code>set_design_top</code>	Specifies the association between the 1801 file and the design top module.
<code>set_domain_supply_net</code>	Specifies the primary power net and the primary ground net for a power domain. Every power domain must have these supply nets defined.
<code>set_isolation</code>	Specifies a power domain (with the <code>-domain</code> option) and the ports on the power domain to isolate using the specified strategy. If an SV-UDN net is used in the isolation rule with the <code>-no_isolation</code> option, the tool disables the WUDNPFS error and proceeds with processing.

<code>set_isolation_control</code>	In the 1801-2013 specification, the <code>set_isolation_control</code> command was deprecated. With Xcelium, if the <code>set_isolation_control</code> command is specified, the tool produces a LGCYWRN warning and all values are passed internally to the <code>set_isolation</code> command.
<code>set_port_attributes</code>	Specifies characteristics (or attributes) to describe ports on a power domain interface. These attributes define a port's isolation, and power and ground requirements. If <code>set_port_attributes</code> command is specified in a design where the top-level port is struct-based SV-UDN, the tool ignores the command and continues processing. See set_port_attributes in the <i>Low-Power Simulation Guide (IEEE 1801)</i>.
<code>set_scope</code>	Changes the current scope within an 1801 power intent file.
<code>set_retention</code>	specifying the supply sets or nets used to power the logic defined by the corresponding retention strategy.
<code>set_retention_control</code>	Specifies a power domain (with the <code>-domain</code> option), the objects in the domain that need to be retention registers, and the save and restore signals and conditions for the retention functionality.
<code>set_related_supply_net</code>	Specifies default power and ground supplies on design objects. It is an extension to the UPF 1.0 specification.
<code>set_simstate_behavior</code>	Specifies the simulation simstate behavior for a model or an instance.
<code>supply_on</code>	Changes the power state of the specified supply port or supply net to ON and updates the voltage of the supply.

Related Topics

- [IEEE 1801 Support in Xcelium](#)
- [Using Common Power Format in Mixed-Signal Designs](#)

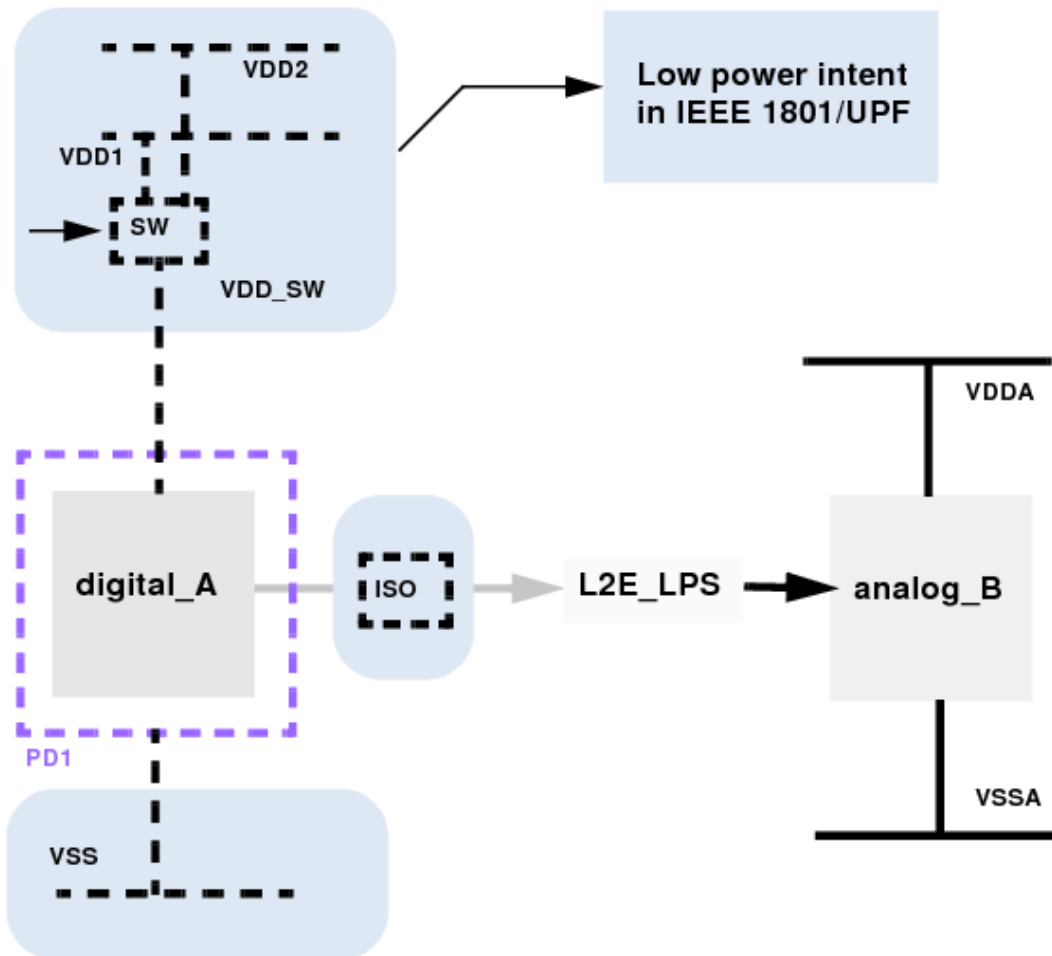
Power Smart Connect Modules for IEEE 1801

The Xcelium simulator mixed-signal option supports the same power-smart connect modules for the IEEE 1801 standard that are supported for CPF. The names of the power-smart connect modules have been unified to have the suffix `_LPS`.

The low-power intent is described in the IEEE 1801 file, and the 1801 flow is enabled by adding the `-lps_1801` option to the `xrun` or `xmelab` command line. The selection and insertion of power-smart connect modules complies with the Verilog-AMS language semantics.

The power-smart connect modules parameter `vpso` is also supported with IEEE 1801 in the `ie` card configuration. This parameter is used to specify the voltage value passed to the analog block when the digital block driving it is shut off. The default value of `vpso` is 0.2.

The following figure illustrates how a power-smart connect module reflects the digital block's low power intent in IEEE 1801 file, and propagates to analog block:



In the above figure, the low-power intent for the digital block `digital_A` is described in IEEE 1801 format and is applied using the `-lps_1801` option. In this case, the power-smart connect module can

identify:

- The power domain state of the `digital_A` block and use the voltage value of the supply set associated with the power domain `PD1` to perform digital-to-analog transition.
- The isolation rule applied on the digital port and propagate the correct analog value to the `analog_B` block.

Refer to the connect library in the installation to see the power-smart connect module definitions.

Related Topic

- [Power-Smart Connect Modules for CPF](#)

Specifying Power Intent on Mixed-Signal Blocks

To describe the power intent for the mixed-signal blocks using the IEEE 1801 format, use the `-lps_ams_sim` option at the `xrun` command line. Based on the power intent information, the built-in power-smart connect modules can be inserted at the mixed-signal boundary. The built-in power-smart connect modules:

- Propagate the power state over the mixed-signal boundary
- Monitor the working voltage conflicts of multiple drivers and loads
- Check the voltage compatibility between the driver side and load side

Power corruption is applied to the wreal signals in power-shutoff state and is controlled by the following options:

- `default`: Forces the wreal signal to ``wrealXState` during the power-shutoff state
- `-lps_wreal_nocorrupt`: Does not apply a corruption value on the wreal signal during power-shutoff state
- `-lps_wreal_corrupt_value <value>`: Applies the specified value on the wreal signal during power-shutoff state. The value can be a real number, `'wrealXState'/'x'` or `'wrealZState'/'z'`

Additionally, you can use the AMSCCPF VPI task, `$amslps_power_state_register(iPwrState, vPwrArgArray)`, to define your own power-aware connect module. This VPI task could be used to get the following information:

- the power state, by back tracing the drivers of input port of smart-ie

- the nominal voltages, which are specified in the CPF control file
- the power shut-off voltage on analog side

Where:

- `iPwrState` is a register, which is associated with power state of driver side (it could be logic, wreal or electrical), and its value is updated automatically if power state changes happen during simulation.
 - `1'b1`: driver is in power on state
 - `1'b0`: driver is in power off state
 - `1'bX`: driver is in standby state
 - `1'bZ`: driver is in power transition
- For Bidir, the `iPwrState` is a register vector `[1:0]iPwrState`. `iPwrState[0]` is used to track power state of digital driver and `iPwrState[1]` is used to track power state of analog side.
- `vPwrArgArray` is a real array variable, which is associated with power domain working voltage info, on digital side, that specified in LP(CPF or IEEE1801) files. And the value is changed automatically if power mode changed during simulation.
 - `vPwrArgArray[0]`: power corruption value of electrical signal
 - `vPwrArgArray[1]`: supply voltage of power domain
 - `vPwrArgArray[2]`: ground voltage of power domain
 - `vPwrArgArray[3]`: pmos bias voltage of power domain
 - `vPwrArgArray[4]`: nmos bias voltage of power domain
 - `vPwrArgArray[5]`: power corruption value of wreal signal

By default, the `amslps-vpi` task links the real array with voltage info of the digital side. However, you can use the `-lps_ams_avref` option to access the power supply of the power-smart connect modules from the power domain of the analog side.

Related Topic

- [Power-Aware Analog/RNM Blocks](#)

Power-Aware Analog/RNM Blocks

Support for the IEEE 1801 standard is available on a mixed-signal network in which a low power-aware signal path is correctly interpreted between the digital and analog driver/receiver, and a low power-aware supply connection is correctly established to connect both analog and digital power supplies.

The `connect_supply_net` command in IEEE 1801 has been enhanced to support the connection of mixed signal network.

```
connect_supply_net net_name -ports { port_list }
```

Signals in `port_list` with type `logic` and `supply_net_type` are handled by UPF and AMS-LPS processing ignores these signals automatically.

UPF supply nets can be connected to design supply nets such as electrical, wreal, and SV-Real Number Modeling (SV-RNM) nets using the `connect_supply_net` command. This is a unidirectional connection, where the voltage info of the UPF supply nets propagates to design supply nets; and, the power state info on the UPF supply nets are ignored. You can enable this feature using the `xrun` option, `-lpa_ams_uca` at the command line.

Additionally, the UPF supply nets and design supply nets can be bidirectionally connected using the `xrun` option, `-lps_ams_connect_supply`. The bidirectional connect module, `ER_bidir` is automatically inserted on the boundary of UPF and design. This allows the voltage info of the UPF supply nets to propagate to the design supply nets.

For AMSD LP1801 Test Bench, you must not enable both `-lps_ams_connect_supply` and `-lps_ams_uca` in the same simulation. Cadence recommends you use the `-lps_ams_connect_supply` command instead of `-lps_ams_uca`.

In the bidirectional connection, the IEEE 1801 command `create_hdl2upf_vct` converts the analog voltage in the design to IEEE 1801-defined supply net value and state. Low power-aware connect modules are now aware of the dynamic power supply and convert the signals from logic to electrical accordingly. For example:

```
create_hdl2upf_vct VCTR2U -hdl_type {sv cds_rnm} -table { {Expression PowerNetState} ... }  
connect_supply_net SUPPLY_NET -ports { powr_list } -vct VCTR2U
```

Here, `cds_rnm` is a reserved keyword to allow real number content in the `-table` argument. It indicates the mapping of real-valued expressions to the IEEE 1801 supply nets for lower power simulation.

The datatypes for SV-UDN can be real (`real`, `realtime`), logic (`bit`, `logic`, `reg`), integer (`int`) and `enum`. And, you need to specify the element of SV-UDN structure. For example:

```
create_hdl2upf_vct vct_name -hdl_type { sv UDN.RealElement } -table {{real_expression  
state_value}*}
```

The expression contains an operator and a value. Following operators are supported:

- Relational: > , >= , < , and <=
- Logical Equality: == and !=
- Case Equality: === and !==

The value can be a real number, ``wrealXState`, or ``wrealZState`. The case equality operators can contain only ``wrealXState`, or ``wrealZState` as values. Variables or parameters are not supported in HDL code.

If there is no VCT specified for the conversion from Wreal to UPF supply voltage, the tool maps the ``wrealZState`wrealXState` to '-1' microvolt.

Mappings with == and === take the highest priority, whereas mappings with != and !== take the lowest priority. In addition, mappings with the greatest threshold for > or >= or lowest threshold for < or <= take the highest priority.

UPF supply nets must have state and voltage. The UPF state can be UNDETERMINED, PARTIAL_ON, FULL_ON, OFF.

wreal and electrical ports computed by RNM or Verilog-AMS or Spectre/SPICE model provide only the voltage and no UPF state. When such port drives a UPF supply net, a UPF VCT (Value Conversion Table) must be defined and set at the connection definition. The VCT creates the UPF states for the UPF supply net.

The following example illustrates that when an LDO instance is simulated (in Spectre or SPICE), the LDO output, named `vout11` in SPICE sub-circuit, drives a UPF supply net named `VDD11sn`:

```
# CREATE A UPF SUPPLY NET
create_supply_net VDD11sn
# DEFINE VALUE CONVERSION TABLE CONVERTING REAL VALUES INTO net_state_type UPF VALUES
create_hdl2upf_vct VCTR2U_VDD11 -hdl_type {sv cds_rnm} \-table { {>=1.0 FULL_ON} {>=0.5 PARTIAL_ON}
{<0.5 OFF} }
# IT IMPORTANT TO ATTACHED THE VCT ONLY TO SUCH CONNECT STATEMENT BECAUSE IT TOUCHED A TRUE SPICE
ELECTRICAL PORT
connect_supply_net VDD11sn -ports {I0_top_ana/LDO_SPICE/vout11} -vct VCTR2U_VDD11
# THE RESULT WILL BE THE FOLLOWING
# WHEN V(I0_top_ana/LDO_SPICE/vout11) > 1.0 THEN the UPF VDD11sn WILL BE IN "FULL_ON" state.
# ELSE IF V(I0_top_ana/LDO_SPICE/vout11) >=0.5 THEN the UPF VDD11sn WILL BE IN "PARTIAL_ON" state
# ELSE IF V(I0_top_ana/LDO_SPICE/vout11) <0.5 THEN the UPF VDD11sn WILL BE IN "OFF" state.
```

Also, the power state info on the UPF supply nets can be converted to an analog voltage in design using the IEEE 1801 command, `create_upf2hdl_vct`, has also been enhanced to describe the mapping from the power net to the voltage gate value, as shown in the following example:

```
create_upf2hdl_vct VCTU2RGND -hdl_type {sv cds_rnm} -table {{OFF `wrealZstate}
{UNDETERMINED `wrealXState} {FULL_ON 0}}
```

```
connect_supply_net GND -ports { powr_list } -vct VCTU2RGND
```

- If the power state is `OFF`, the simulator applies a value ``wrealZState` on the mixed-signal supply net;
- If the power state is `UNDETERMINED`, the simulator applies a value ``wrealXState` on the mixed-signal supply net;
- If the power state is `FULL_ON`, the simulator applies a value `0` on the mixed-signal supply net.

For VCT definitions in UPF for SV-UDN connections, you need to specify the element of SV-UDN structure. For example:

```
create_hdl2upf_vct vct_name -hdl_type { sv UDN.RealElement } -table {{real_expression  
state_value}*}
```

The datatypes for SV-UDN can be real (real, realtime), logic (bit, logic, reg), integer (int) and enum. And, you can apply VCTs on multiple SV-UDN fields with same datatype. For example:

```
create_upf2hdl_vct VCTS2V -hdl_type {sv EEnet.V} \  
-table { {FULL_ON 1.5} {OFF 0.0} }  
create_upf2hdl_vct VCTS2I -hdl_type {sv EEnet.I} \  
-table { {FULL_ON 0.001} {OFF 0.0001} }  
create_upf2hdl_vct VCTS2R -hdl_type {sv EEnet.R} \  
-table { {FULL_ON 100} {OFF 50} }  
  
connect_supply_net vdd -ports {i_ana2/vdd} -vct VCTS2V  
connect_supply_net vdd -ports {i_ana2/vdd} -vct VCTS2I  
connect_supply_net vdd -ports {i_ana2/vdd} -vct VCTS2R
```

If there is a supply connection specified in Verilog model, the tool applies a connection between the RTL and UPF signal. If the same port name exists in UPF and HDL models, a connection is established on the UPF supply port.

In the following example, tool applies a connection between UPF supply net `VDD` and HDL net `ipmu/VDD`.

```

-----tb.upf-----
load_upf top.upf -scope dut
create_supply_net VDD
connect_supply_net VDD -ports {dut/VDD} // connecting to UPF
connect_supply_net VDD -ports {ipmu/VDD} // connecting to HDL
-----top.upf-----
create_supply_port VDD
create_supply_net VDD
connect_supply_net VDD -ports {VDD} // connecting to UPF
connect_supply_net VDD -ports {iainv/VDD} // connecting to HDL
// optimized away since the HDL signal "iainv/VDD" is
connecting to "ipmu/VDD" in HDL scope.
connect_supply_net VDD -ports {iwinv/VDD} // connecting to HDL
// optimized away since the HDL signal "iwinv/VDD" is
connecting to "ipmu/VDD" in HDL scope.
-----tb.sv-----
module tb();
    pmu ipmu(VDD);
    top dut(VDD);
endmodule;
-----top.vams-----
module top(VDD);
    ainv iainv(VDD);
    winv iwinv(VDD);
endmodule
    
```

Additionally, SV-UDN connections can be used in IEEE1801 power supply connection with Liberty model. The datatypes for SV-UDN can be real (real, realtime), logic (bit, logic, reg), integer (int) and enum.

As SV-UDN has multiple fields, you must specify multiple VCTs on same UDN net, to specify the value conversion for power state and voltage.

The following is an example of the HDL source, VCT definitions for the UPF supply nets, and the liberty model. The tool applies the supply connection on SV-UDN net automatically based on the UPF description. The specified VCT is used for value conversion of the specified field.

```

-----HDL Source Code-----
module dut();
    anal i_anal();
endmodule

module anal(vdd, vss);
    output wTsum vdd, vss;
end
endmodule

-----UPF Definition-----
connect_supply_net vdd -ports {i_anal/vdd} -vct VCTR2V
connect_supply_net vdd -ports {i_anal/vdd} -vct VCTR2S_VDD
connect_supply_net vss -ports {i_anal/vss} -vct VCTR2V
    
```

```
connect_supply_net vss -ports {i_anal/vss} -vct VCTR2S_VSS

-----Liberty Model-----
library ("AMSLPS_TEST") {
cell ("anal") {
pg_pin (vss) {
pg_type : "primary_ground";
voltage_name : "VSS";
}
pg_pin (vdd) {
pg_type : "primary_power";
voltage_name : "VDD";
}
}
}

-----Elaboration Result-----
Liberty Verbose Information

anal : ana.lib:88
<PA>: ./dut.sv:8
Instances:
tb.i_dut.i_anal
<E> vss <primary_ground> : vss <VCTR2S_VSS>
<E> vss <primary_ground> : vss <VCTR2V>
<E> vdd <primary_power> : vdd <VCTR2S_VDD>
<E> vdd <primary_power> : vdd <VCTR2V>

Liberty / HDL Matching Information
No mismatch found between Liberty pins and corresponding Verilog ports.
```

Specifying Parameter Values for Connect Modules on Power Supply Nets

To specify an ideal voltage source for the connect module on power supply nets, you can use the `idealmode` parameter that is part of the built-in E-R connect module (`ER_bidir`). The parameter can be set in the following two ways:

- **Using the `ie` statement:** You can set the value of the `route` parameter to 0, which changes the value of `idealmode` parameter to 1 automatically.
- **Using the `-amsconrules` option:** You can specify the value of the `idealmode` parameter in the user-defined connect rules.

By default, the power supply net takes the discipline `discrete_amslps`. However, you can set the discipline for a particular power supply net by specifying the `-setdiscipline` option at the command

line. The tool automatically selects the connect module from the connect rules based on the discipline of IEEE1801 power supply net.

```
# CREATE A UPF SUPPLY NET
create_supply_net VDD11sn
# DEFINE VALUE CONVERSION TABLE CONVERTING REAL VALUES INTO net_state_type UPF VALUES
create_hdl2upf_vct VCTR2U_VDD11 -hdl_type {sv cds_rnm} \

-table { {>=1.0 FULL_ON} {>=0.5 PARTIAL_ON} {<0.5 OFF} }
# IT IS IMPORTANT TO ATTACH THE VCT ONLY TO SUCH CONNECT STATEMENT BECAUSE IT TOUCHES A
TRUE SPICE ELECTRICAL PORT
connect_supply_net VDD11sn -ports {I0_top_ana/LDO_SPICE/vout11} -vct VCTR2U_VDD11
# THE RESULT WILL BE THE FOLLOWING
# WHEN V(I0_top_ana/LDO_SPICE/vout11) > 1.0 THEN the UPF VDD11sn WILL BE IN "FULL_ON"
state.
# ELSE IF V(I0_top_ana/LDO_SPICE/vout11) >=0.5 THEN the UPF VDD11sn WILL BE IN
"PARTIAL_ON" state
# ELSE IF V(I0_top_ana/LDO_SPICE/vout11) <0.5 THEN the UPF VDD11sn WILL BE IN "OFF" state.
```

Related Topic

- [Power Smart Connect Modules for IEEE 1801](#)

Automatic Supply Connections in Mixed-Signal Designs with Liberty Model

In mixed-signal designs, when connecting UPF supply net with analog (electrical/wreal) signal, a VCT needs to be specified along with the `connect_supply_net` command for the value conversion between UPF power state and HDL supply voltage. And, the tool automatically, creates a connect supply net between HDL and UPF based on the specified VCT.

However, in mixed-signal designs with Liberty models, you can enable automatic supply connections in designs with higher HDL connections and default VCT.

- [Enabling Automatic Supply Connection in Designs with Higher HDL Connections](#)
- [Enabling Automatic Supply Connection in Designs Using Liberty Models with a Default VCT Value](#)

Enabling Automatic Supply Connection in Designs with Higher HDL Connections

If there is a mixed-signal supply network existing on both HDL and UPF scope; and, if the supply nets have the same name with UPF supply nets on the design top, use the `-lps_ams_supply_hiconn` command-line option to instruct the tool to trace the HDL high connection of power ports of liberty model, and optimize the automatic supply connection between HDL and UPF to the top-level design. The VCT of this connection is used either from the explicit UPF command `connect_supply_net` on port ports of liberty model or the default VCT command `-lps_ams_default_power_vct/-lps_ams_default_ground_vct`.

For value conversion between the analog voltage and UPF power state, you must specify the default analog VCT through `xrun` command-line option, `connect_supply_net`. The tool then uses the default VCT based on connect direction for analog value conversion. If the default VCT is not specified, the tool ignores the connection on this net. Then, you must explicitly specify the VCT using the `connect_supply_net` command in the UPF file for that particular net. The only exception when you do not have to specify VCT is when the signal is electrical/wreal. In this case, the tool propagates voltage values from UPF to HDL directly without any VCT and applies an automatic connection.

The following are some design scenarios and how the tool behaves when the `-lps_ams_supply_hiconn` option is specified for the automatic supply network:

- If there is no UPF supply net defined with the same name at the top level, the tool automatically creates an internal reference UPF supply net for supply connection and reports a warning.
- If a liberty model is encountered and its supply port is treated as analog (electrical, wreal, SV-UDN), the tool traces the high connection of the supply port and gets the HDL supply net on the highest hierarchical level. If the HDL supply net has the same as UPF supply net, then the tool applies a connection between that HDL supply net and the UPF supply net on the top level, instead of the supply port of liberty model on a leaf.
- If the net at the top level is logic, the tool reports a warning and ignores the connection on this net.
- If the selected UPF supply net at the top level (coming from HDL high connection) is different from the original one (coming from UPF definition), the UPF supply net at the top level from HDL connection is used for automatic supply connection. However, for such cases, you can explicitly specify the `connect_supply_net` command in the UPF file to connect the supply net.
- If the analog net has the same name as that of the UPF supply net, the tool does not create an

automatic supply connection on it and reports a warning. If you want the tool to connect these supply nets, you can specify using the `connect_supply_net` command in the UPF file.

If you have specified the `connect_supply_net` option with VCT, it takes priority and overwrites the `-lps_ams_supply_hiconn` command-line option.

Additionally, in mixed-signal designs if you want the tool to use an existing UPF supply specified in the UPF file, you can use the `-lps_infer_upf_supply` command-line option. If the tool identifies a connection on the specific signal in HDL that has explicit supply connection to another UPF supply net, the tool redirects the supply connection on this signal to this specific UPF supply net specified in the UPF file.

In the following example, the tool connects `tb.dut.i_ana1.vdd` to supply net `vdd` instead of supply net `vdd1`, which is part of supply set `SSD` and power domain `PDD`.

```
-----Top.sv-----
module top();
wire avdd, avss;
ana1 i_ana1(avdd, avss);
ana2 i_ana2(avdd, avss);
endmodule

module ana1(vdd, vss);
output wTsum vdd, vss;
endmodule

module ana2(vdd, vss);
input wTsum vdd, vss;
endmodule

-----Top.upf-----
create_supply_net vdd
create_supply_net vss
create_supply_net vdd1
create_supply_net vss1
create_supply_set SSD -function {power vdd1} -function {ground vss1}
create_power_domain PDD -supply {primary SSD} -include_scope
connect_supply_net vdd -ports {i_ana2/vdd}
connect_supply_net vss -ports {i_ana2/vss}
```

And, the tool reports the following MPSNIF warning message:

```
xmelab: *W,MPSNIF: [LPS] The program redirects the supply connection of signal
(tb.dut.i_ana1.vss) from UPF supply net (vss1) to (vss).
xmelab: *W,MPSNIF: [LPS] The program redirects the supply connection of signal
(tb.dut.i_ana1.vdd) from UPF supply net (vdd1) to (vdd).
```

Related Topics

- [Automatic Supply Connections in Mixed-Signal Designs with Liberty Model](#)
- [Enabling Automatic Connection in Designs Using Liberty Models with a Default VCT Value](#)

Enabling Automatic Supply Connection in Designs Using Liberty Models with a Default VCT Value

In mixed-signal designs with UPF, the tool cannot apply an automatic supply connection on the analog ports, based on the Liberty model definition because there is no VCT explicitly specified on these analog ports. In such cases, to specify the default VCT for automatic power supply connection on the analog ports with UPF supply nets, use the following two `xrun` command-line options:

- `-lps_ams_default_power_vct <VCT_for_Power_Net>`
- `-lps_ams_default_ground_vct <VCT_for_Ground_Net>`

If a liberty model is encountered and its supply pin is treated as analog (electrical, wreal or UDN), the tool applies the specified default VCT on this supply connection.

You can specify multiple default VCTs using these two options. The first VCT that adheres to all the connection criteria is used for automatic supply connection. Similarly, when you specify multiple VCTs for different nettype and UDN fields, the tool uses the first nettype or UDN field that matches with the analog net.

The P/G type of supply connection is determined by the `-pg_type` value specified in the Liberty model file. The tool uses the VCT from the default VCT list specified for power or ground pins:

- For power pins, the tool uses the VCT from the list of options specified in the `-lps_ams_default_power_vct` option.
- For ground pins, the tool uses the VCT from the list of options specified in the `-lps_ams_default_ground_vct` option.

The direction of the supply connection is determined by the port direction specified in the Liberty and HDL models. The tool uses the VCT based on the direction of VCT definition such as:

- For the UPF driving HDL connection, the tool uses the VCT created by the UPF command, `create_upf2hdl_vct`.
- For the HDL driving UPF connection, the tool uses the VCT created by the UPF command, `create_hdl2upf_vct`.

The hdl type of VCT definition should meet the nettype of analog signals, such as

- The VCT defined with `-hdl_type {sv cds_rnm}` can be used for electrical or wreal signals.
- The VCT defined with `-hdl_type {sv UDN.field}` can be used for UDN signals.

For SV-UDN connections, the tool applies multiple VCTs, if they are defined for the same UDN nettype but different fields.

- You cannot specify default VCTs on a specific scope of a design; that is, you cannot specify a set of VCTs on one block and another set of VCTs on other blocks.
- The `-connect_supply_net` takes higher priority if a supply connection is explicitly specified in UPF file.

The following example illustrates how to define the VCT in the UPF definition and how to provide the VCT values for HDL to UPF connections and vice-versa using the `-lps_ams_default_power_vct` and `-lps_ams_default_ground_vct` options.

UPF Definition:

```
create_hdl2upf_vct VCTR2U_VDD -hdl_type {sv cds_rnm} \
    -table { {>=1.0 FULL_ON} {<=0.0 OFF} }
create_hdl2upf_vct VCTR2U_VSS -hdl_type {sv cds_rnm} \
    -table { {>=0.0 FULL_ON} {<0.0 OFF} }
create_upf2hdl_vct VCTU2R -hdl_type {sv cds_rnm} \
    -table { {UNDETERMINED `wrealXState} }
```

xrun Command:

```
# HDL->UPF
-lps_ams_default_power_vct VCTR2U_VDD
-lps_ams_default_ground_vct VCTR2U_VSS
# UPF->HDL
-lps_ams_default_power_vct VCTU2R
-lps_ams_default_ground_vct VCTU2R
```

Liberty Verbose Information:

```
anal : ana.lib:88
<PA>: ./dut.vams:6
Instances:
  tb.i_anal
    <A> vss <primary_ground> : Global_Always_On_Ground <VCTR2U_VSS>
    <A> vdd <primary_power> : Global_Always_On_Power <VCTR2U_VDD>
  tb.i_dut.i_anal
    <A> vss <primary_ground> : vss <VCTR2U_VSS>
    <A> vdd <primary_power> : vdd <VCTR2U_VDD>
```

Liberty / HDL Matching Information

No mismatch found between Liberty pins and corresponding Verilog ports.

ana2 : ana.lib:102

<PA>: ./dut.vams:23

Instances:

tb.i_ana2

<A> vss <primary_ground> : Global_Always_On_Ground <VCTU2R>

<A> vdd <primary_power> : Global_Always_On_Power <VCTU2R>

tb.i_dut.i_ana2

<A> vss <primary_ground> : vss <VCTU2R>

<A> vdd <primary_power> : vdd <VCTU2R>

Liberty / HDL Matching Information

No mismatch found between Liberty pins and corresponding Verilog ports.

Enabling UPF Supply Connection on Mixed-Signal Designs Using SV-AMS Connect Modules

You can enable UPF supply connections in mixed-signal designs that use SV-AMS connect modules. The mixed signal supply net could be electrical, SV-UDN, SV-Wreal, and VAMS-wreal with different nettypes or resolution functions. The tool inserts different connect modules for each nettypes on same UPF supply net.

To enable this functionality, you must use the command-line option, `-lps_ams_supply_ie` along with the following options:

- `-lps_ams_connect_supply`: To enable the power supply net connectivity
- `-rnm_dmsie` and `-auto_config_svams_ie`: To enable the SVAMS insertion

To connect UPF supply nets to SV UDN (EEnet), SV wreal (wrealsum, wrealavg), or electrical signals in the RNM/analog blocks, Cadence provides a solution to connect using `cds_ms_upf_supply` nettype defined in SV package `CDS_MS_UPF`. The SV package is created under Xcelium installation path, `your_install_dir/tools.lnx86/affirma_ams/etc/lpms/cds_ms_upf_package.sv`, where a predefined MS-UPF nettype is included.

```
package CDS_MS_UPF;

typedef struct {
  state state;
  real voltage;
} cds_ms_upf_supply_struct;
```

During elaboration, an internal SV-UDN net `VDD_MS` (with nettype `cds_ms_upf_supply`) is created to represent the UPF supply net `VDD_UPF` in HDL scope. The tool connects all the mixed signals that is

connecting to same UPF supply net `VDD_UPF` in UPF definition, to the SV-UDN net `VDD_MS`, using SVAMS connect module insertion.

You may also refer to the built-in connect modules: `UPF_EEnet.svams`, `UPF_Wreal.svams`, and `UPF_Electrical.svams`. These built-in connect modules are available at the Xcelium installation path, `your_install_dir/tools.lnx86/affirma_ams/etc/lpms`.

During simulation, the tool propagates the values between SV-UDN, SV wreal, and electrical nets; the internal SV-UDN net `VDD_MS`; and the UPF supply net `VDD_UPF`.

In the UPF specification, the analog VCT is not required on the explicit supply connection, if the `-lps_ams_supply_ie` option is specified.

If you want the tool to use custom MS-UPF supply nettype, you must specify a bidirectional connect module between the mixed signal nettype and user-defined MS-UPF nettype. In this case, you must use the following two additional options to specify the nets:

- `-lps_ams_supply_nettype <user-defined UPF supply nettype>`: To specify the user defined nettype MS-UPF to represent UPF supply net for mixed signal connection.
- `-lps_ams_supply_vctfile <user-defined UPF vct>`: To specify the analog VCT definition for the value conversion between user defined nettype MS-UPF and UPF supply nettype. See [Example 6](#).

Additionally, if you have multiple drivers, you can use the `-lps_ams_supply_driver <net name>` option to specify the net as a driver of UPF supply net.

Since the SV-UDN has multiple fields, you must specify multiple VCTs on same UDN net, to specify the value conversion for power state and voltage. The tool relaxes the error checking (*E,DUPGCN) on SV-UDN nets, and apply the VCTs on specified fields of UDN net.

You can also use ie card to specify the IE parameters for different UPF supply nets. The connect modules for different nettypes connecting to same UPF supply net shares the same ie card info. Ensure that you specify the full hierarchical path UPF supply net name. See [Example 5](#).

```
amsd {  
ie vsup=3.3  
ie vsup=1.8 net=top.VDD vthi=1.6 vtlo=0.5 state_threshold=1  
ie vsup=0.0 net=top.VSS vthi=0.0 state_threshold=1  
}
```

Currently, the following scenarios are not supported:

- The CPF format
- VHDL model

Examples

- [Example 1: Supply connection on SV-UDN net \(EEnet\), SV wreal, Electrical](#)
- [Example 2: Supply connection with multiple drivers, resolved by mixed signal resolution function](#)
- [Example 3: Supply connection with multiple driver, resolved by UPF resolution method](#)
- [Example 4: Liberty model](#)
- [Example 5: ie card specification](#)
- [Example 6: User defined MS-UPF nettype and value conversion](#)

Example 1: Supply connection on SV-UDN net (EEnet), SV wreal, Electrical

```
-----Xrun command args-----
$CDS_INST_DIR/tools/affirma_ams/etc/lpms/UPF_EEnet.svams
$CDS_INST_DIR/tools/affirma_ams/etc/lpms/UPF_Wreal.svams
$CDS_INST_DIR/tools/affirma_ams/etc/lpms/UPF_Electrical.svams

-lps_ams_connect_supply -lps_ams_sim
-lps_ams_supply_ie
-rnm_dmsie -auto_config_svams_ie
-----HDL Design source code-----
`timescale 1ns / 100ps
import EE_pkg::*;
import CDS_MS_UPF::*;

module top ();
  (* upf_always_on *)
  reg pso = 0;
  initial begin
    #200 pso=1;
    #200 pso=0;
  end
  drvr_eenet idrvr_eenet ();
  drvr_wreal idrvr_wreal ();
  drvr_elec idrvr_elec ();

endmodule

module drvr_eenet (vdd, vss);
  output vdd, vss;
  EEnet vdd, vss;
  real v1, v2, r;
```

```

assign vdd = '{V:5.0, I:`wrealZState, R:r};
assign vss = '{V:0.0, I:`wrealZState, R:r};

endmodule

module drvr_wreal (rvdd, rvss);
output rvdd, rvss;
wrealavg rvdd, rvss;
real vsup, vgnd;
assign rvdd = vsup;
assign rvss = vgnd;
endmodule

### test.vams ###
`include "disciplines.vams"
`timescale 1ns / 100ps

module drvr_elec (avdd, avss);
output avdd, avss;
electrical avdd, avss, gnd;
ground gnd;
real vsup=0.0;
vsource #(.dc(0.0), .type("dc")) vgnd (avss, gnd);
analog begin
V(avdd) <+ transition(vsup, 0.0, 1e-9, 1e-9, 0.0);
end
endmodule

-----UPF Description-----
upf_version 2.0

set_design_top top
set_scope

create_supply_net VDD
create_supply_net VSS

create_supply_net VDD1
create_supply_net VSS1

create_supply_set SSD -function {power VDD} -function {ground VSS}
create_supply_set SS1 -function {power VDD1} -function {ground VSS1}

create_power_domain PDD -supply {primary SSD} -include_scope
    
```

```
create_power_domain PD1 -supply {primary SS1} -elements {iload}

connect_supply_net VDD -ports {idrvr_eeenet/vdd idrvr_wreal/rvdd idrvr_elec/avdd }
connect_supply_net VSS -ports {idrvr_eeenet/vss idrvr_wreal/rvss idrvr_elec/avss}
```

The following is the ieinfo log information:

```
Interface Elements at the block <instance> cds_amslps_psnconn of <master>
cds_amslps_psnconn (file : ./xcelium.d/AMSD/cds_amslps_simulation.sv)
Automatically inserted : top.VDD__EEenetUPF_bidir__ddiscrete_1_8
Connect Module : EEenetUPF_bidir
Mode : Merged
Net : top.VDD (discipline: ddiscrete_1_8, nettype: cds_ms_upf_supply)
Parameters :
rlo :200
rhi :200
rx :40
rz :10000000
```

```
List of Ports connected to net top.VDD : (Total: 1)
top.idrvr.vdd (<none> output)
```

```
Discipline of Port (e): No Discipline
Discipline of Port (u): No Discipline
```

```
Sensitivity information:
No Sensitivity info
```

Example 2: Supply connection with multiple drivers, resolved by mixed signal resolution function

```
-----HDL Design source code-----
### top.sv ###
`timescale 1ns / 100ps

import EE_pkg::*;
import CDS_MS_UPF::*;

module top ();
  drvvr_udn idrvr_udn ();
  drvvr_ele idrvr_ele ();
  drvvr_rnm idrvr_rnm ();
  load_udn iload_udn ();
```



```
load_ele iload_ele ();
load_rnm iload_rnm ();
endmodule

module drvr_udn (vdd, vss);
output vdd, vss;
EEnet vdd, vss;
real v1, v2, r;

assign vdd = '{V:v1, I:`wrealZState, R:r};
assign vss = '{V:v2, I:`wrealZState, R:r};

endmodule

module load_udn (vdd, vss);
inout vdd, vss;
EEnet vdd, vss;
EEnet evdd, evss;
assign evdd = vdd;
assign evss = vss;
endmodule

### test.vams ###
`include "disciplines.vams"
`timescale 1ns / 100ps

module drvr_ele (avdd, avss);
output avdd, avss;
electrical avdd, avss, gnd;
ground gnd;
real vsup=0.0;
real rout = 1e12;
real RealA;
real iAout;
endmodule

module drvr_rnm (avdd, avss);
output avdd, avss;
wrealavg avdd, avss;
real vsup, vgnd;
assign avdd = vsup;
assign avss = vgnd;

endmodule
```

```

module load_ele (avdd, avss);
inout avdd, avss;
electrical avdd, avss;
endmodule

module load_rnm (avdd, avss);
inout avdd, avss;
wrealavg avdd, avss;
endmodule

-----UPF Description-----
upf_version 2.0
set_design_top top
set_scope

create_supply_resolution_function cust -voltage high -state most_on

create_supply_net VDD -resolve cust
create_supply_net VSS -resolve cust

create_supply_set SSD -function {power VDD} -function {ground VSS}
create_power_domain PDD -supply {primary SSD} -include_scope

connect_supply_net VDD -ports {idrvr_udn/vdd idrvr_ele/avdd idrvr_rnm/avdd}
connect_supply_net VSS -ports {idrvr_udn/vss idrvr_ele/avss idrvr_rnm/avss}
connect_supply_net VDD -ports {iload_udn/vdd iload_ele/avdd iload_rnm/avdd}
connect_supply_net VSS -ports {iload_udn/vss iload_ele/avss iload_rnm/avss}
-----Supply Net Resolution-----
Supply Net Resolution:
[100150 PS]Driving Source:
cds_amslps_psnconn.uVDD_1_11, Voltage: 1.600000V, State FULL_ON
Resolved Net:
top.VDD, Voltage: 1.600000V, State: FULL_ON
    
```

Example 3: Supply connection with multiple driver, resolved by UPF resolution method

```

-----xrun options-----
-lps_ams_supply_driver "top.idrvr_ele.avdd"
-lps_ams_supply_driver "top.idrvr_ele.avss"
-lps_ams_supply_driver "top.idrvr_udn.vdd"
-lps_ams_supply_driver "top.idrvr_udn.vss"
    
```

```
-lps_ams_supply_driver "top.idrvr_rnm.avdd"
-lps_ams_supply_driver "top.idrvr_rnm.avss"
-----HDL Design source code-----

### top.sv ###
`timescale 1ns / 100ps

import EE_pkg::*;
import CDS_MS_UPF::*;

module top ();
  drvvr_udn idrvr_udn ();
  drvvr_ele idrvr_ele ();
  drvvr_rnm idrvr_rnm ();
  load_udn iload_udn ();
  load_ele iload_ele ();
  load_rnm iload_rnm ();
endmodule

module drvvr_udn (vdd, vss);
  output vdd, vss;
  EEenet vdd, vss;
  real v1, v2, r;

  assign vdd = '{V:v1, I:`wrealZState, R:r};
  assign vss = '{V:v2, I:`wrealZState, R:r};

endmodule

module load_udn (vdd, vss);
  inout vdd, vss;
  EEenet vdd, vss;
  EEenet evdd, evss;
  assign evdd = vdd;
  assign evss = vss;
endmodule

### test.vams ###
`include "disciplines.vams"
`timescale 1ns / 100ps

module drvvr_ele (avdd, avss);
  output avdd, avss;
  electrical avdd, avss, gnd;
  ground gnd;
```

```

real vsup=0.0;
real rout = 1e12;
real RealA;
real iAout;

endmodule

module drvr_rnm (avdd, avss);
output avdd, avss;
wrealavg avdd, avss;
real vsup, vgnd;
assign avdd = vsup;
assign avss = vgnd;
endmodule

module load_ele (avdd, avss);
inout avdd, avss;
electrical avdd, avss;
endmodule

module load_rnm (avdd, avss);
inout avdd, avss;
wrealavg avdd, avss;
endmodule

-----UPF Description-----
upf_version 2.0
set_design_top top
set_scope

create_supply_resolution_function cust -voltage high -state most_on

create_supply_net VDD -resolve cust
create_supply_net VSS -resolve cust
create_supply_set SSD -function {power VDD} -function {ground VSS}

create_power_domain PDD -supply {primary SSD} -include_scope

connect_supply_net VDD -ports {idrvr_udn/vdd idrvr_ele/avdd idrvr_rnm/avdd}
connect_supply_net VSS -ports {idrvr_udn/vss idrvr_ele/avss idrvr_rnm/avss}
connect_supply_net VDD -ports {iload_udn/vdd iload_ele/avdd iload_rnm/avdd}
connect_supply_net VSS -ports {iload_udn/vss iload_ele/avss iload_rnm/avss}
-----Supply Net Resolution-----
//The multiple drivers are resolved on UPF side, there are multiple driver connected on
UPF supply net:
Supply Net Resolution:
    
```

```
[100 NS]Driving Source:
cds_amslps_psnconn.uVDD_1_11_DRV1, Voltage: 1.600000V, State FULL_ON
cds_amslps_psnconn.uVDD_1_11_DRV2, Voltage: 0.000000V, State FULL_ON
cds_amslps_psnconn.uVDD_1_11_DRV3, Voltage: 0.000000V, State OFF
Resolved Net:
top.VDD, Voltage: 1.600000V, State: FULL_ON
```

Example 4: Liberty model

```
-----HDL Design source code-----
### top.sv ###
`timescale 1ns / 100ps

import EE_pkg::*;
import CDS_MS_UPF::*;

module top ();
  drvr idrvr ();
  load iload ();
endmodule

### test.scs ###
simulator lang=spectre

subckt drvr avdd avss
X0 (net1 0) vsource type=pwl wave=[0n 0.0 100n 0.0 101ns 1.8 300n 1.8 301n 1.5 500n 1.5
501n 0.0]
X1 (avss 0) vsource dc=0.0
R0 (net1 avdd) resistor r=0.001
C0 (avdd avss) capacitor c=100f
ends drvr

subckt load avdd avss
R0 (avdd net1) resistor r=100k
R1 (net1 avss) resistor r=100k
R2 (avss 0) resistor r=10
ends load

### amscf.scs ###
*
include "test.scs"

amsd {
```

```
portmap subckt=drvvr
config cell=drvvr use=spice
portmap subckt=load
config cell=load use=spice
}

-----Liberty Description-----
cell ("drvvr") {
area : 8546.44;
dont_touch : true;
dont_use : true;
pad_cell : true;
pg_pin (avss) {
pg_type : "primary_ground";
voltage_name : "VSS";
direction : output;
}
pg_pin (avdd) {
pg_type : "primary_power";
voltage_name : "VDD";
direction : output;
}
}
cell ("load") {
area : 8546.44;
dont_touch : true;
dont_use : true;
pad_cell : true;
pg_pin (avss) {
pg_type : "primary_ground";
voltage_name : "VSS";
}
pg_pin (avdd) {
pg_type : "primary_power";
voltage_name : "VDD";
}
}

-----Liberty Verbose Info-----
Liberty Verbose Information
load : test.lib:104
<PA>: ./xcelium.d/AMSD/ams_spice_in/generated_skeleton_amscb.skl_vams:21
Instances:
top.iload
<E> avss <primary_ground> : VSS1
<E> avdd <primary_power> : VDD1
```

Liberty / HDL Matching Information

No mismatch found between Liberty pins and corresponding Verilog ports.

drvvr : test.lib:88

<PA>: ./xcelium.d/AMSD/ams_spice_in/generated_skeleton_amscb.skl_vams:8

Instances:

top.idrvvr

<E> avss <primary_ground> : VSS

<E> avdd <primary_power> : VDD

Liberty / HDL Matching Information

No mismatch found between Liberty pins and corresponding Verilog ports.

Example 5: ie card specification

-----UPF Description-----

upf_version 2.0

set_design_top top

set_scope

create_supply_net VDD

create_supply_net VSS

create_supply_net VDD1

create_supply_net VSS1

create_supply_set SSD -function {power VDD} -function {ground VSS}

create_supply_set SS1 -function {power VDD1} -function {ground VSS1}

create_power_domain PDD -supply {primary SSD} -include_scope

create_power_domain PD1 -supply {primary SS1} -elements {iload}

-----ie Card-----

amsd {

ie vsup=3.3

ie vsup=1.8 net=top.VDD vthi=1.6 vtlo=0.5 state_threshold=1

ie vsup=0.0 net=top.VSS vthi=0.0 state_threshold=1

}

Example 6: User defined MS-UPF nettype and value conversion

```

-----xrun options-----
my_upf_package.sv
my_UPF_*.svams
top.sv
-access rwc -timescale 1ns/100ps
-rnm_dmsie -auto_config_svams_ie
-rnm_info -ieinfo
-lps_1801 top.upf -lps_ams_sim
-lps_ams_connect_supply
-lps_ams_supply_ie
-lps_ams_supply_nettype myUPF_pkg::my_upf_supply
-lps_ams_supply_vctfile my_upf_vct.upf
-input probe.tcl -clean
-----User defined MS-UPF nettype-----
package myUPF_pkg;

typedef struct {
    real V;
    integer driverCnt;
} my_upf_supply_struct;

function automatic my_upf_supply_struct resFunc(input my_upf_supply_struct drivers[]);
my_upf_supply_struct ResolvedValue;
int Count = 0;
real Vol = 0.0;
resFunc.V = `wrealZState;
resFunc.driverCnt = 0;;
foreach (drivers[i]) begin
    if (drivers[i].V === `wrealXState) begin
        ++ Count;
        resFunc.V = `wrealXState;
    end
    else if (drivers[i].V !== `wrealZState) begin
        ++ Count;
        Vol += drivers[i].V;
    end
end
if ((resFunc.V !== `wrealXState) && (Count>0)) begin
    resFunc.V = Vol/Count;
    resFunc.driverCnt = Count;
end

endfunction
    
```



```
nettype my_upf_supply_struct my_upf_supply with resFunc;
endpackage : myUPF_pkg
-----User defined analog VCT-----
# my_upf_vct.upf
create_hdl2upf_vct my_H2UV -hdl_type {sv my_upf_supply.V} \
-table { {===`wrealXState 0.0} {===`wrealZState 0.0} }
create_hdl2upf_vct my_H2US -hdl_type {sv my_upf_supply.V} \
-table { {===`wrealXState UNDETERMINED} {===`wrealZState OFF} {>=0.0 FULL_ON}}
create_upf2hdl_vct my_U2HV -hdl_type {sv my_upf_supply.V} \
-table { {UNDETERMINED `wrealXState} {OFF `wrealZState} }
```

Related Topics

- [Automatic Supply Connections in Mixed-Signal Designs with Liberty Model](#)
- [Enabling Automatic Supply Connection in Designs with Higher HDL Connections](#)

Enabling SystemVerilog Interface in LP-MS Designs

To enable SystemVerilog (SV) interface in low-power mixed-signal designs, use the `lps_sv_interface_port` option. This ensures that the mixed-signal processing complies with the LP SV interface infrastructure:

- AMS discipline resolution processing
- DMS coercion processing
- Connect module net/port database

Related Topics

- [Using the IEEE 1801 Standard in Mixed-Signal Designs](#)
- [Power Smart Connect Modules for IEEE 1801](#)

SystemC Integration for Mixed-Signal Verification

SystemC® is the industry standard for system-level design modeling; it supports the description of the architecture of complex systems consisting of both software and hardware components. The SystemC library provides the constructs necessary to model system architecture, including hardware timing, concurrency, and reactive behavior.

You can simulate and debug a design mixed with SystemC and Verilog or VHDL. Xcelium simulator with mixed-signal option provides support for instantiating Verilog (digital), Verilog-AMS, VHDL (digital), and VHDL-AMS modules inside SystemC models.

Related Topic

- [Instantiating Verilog-AMS and VHDL-AMS in SystemC](#)

Instantiating Verilog-AMS and VHDL-AMS in SystemC

SystemC is a digital-only language. For that reason, the Xcelium simulator with mixed-signal option uses Verilog-AMS wrappers with purely digital ports and SystemC shells to instantiate Verilog-AMS and VHDL-AMS modules within SystemC. Verilog-AMS wrappers are required only if the Verilog-AMS and VHDL-AMS modules contain a mix of analog and digital ports. Instantiations of Verilog-AMS and VHDL-AMS modules (which have purely digital ports already) do not require wrappers and use only SystemC shells. To translate analog signals to and from the digital signals of the SystemC language, Xcelium simulator with mixed-signal option uses connect modules and interface modules.

When using SystemC models with the Xcelium simulator with mixed-signal option, the following restrictions apply:

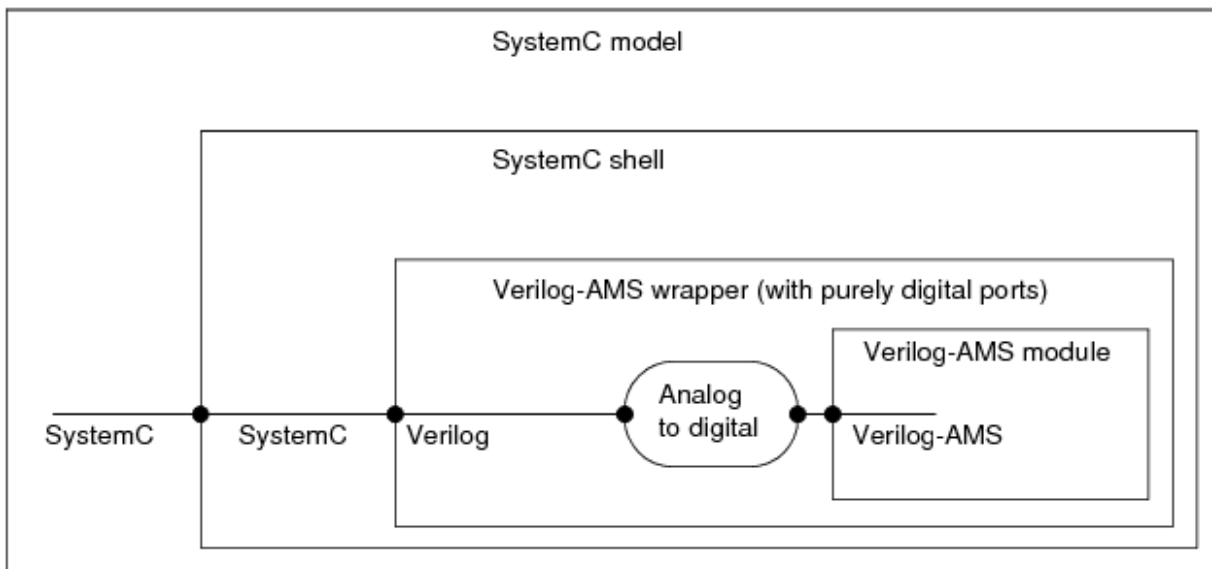
- You must not instantiate a Verilog-AMS module that instantiates a VHDL-AMS module
- You must not instantiate a VHDL-AMS module that instantiates a Verilog-AMS module

See the following topics for more information:

- [Preparing and Using Wrappers and Shells](#)
- [Preparing Interface Modules](#)
- [Guidelines for Using AMS Modules in SystemC Models](#)

Preparing and Using Wrappers and Shells

As the following diagram illustrates, the Xcelium simulator with mixed-signal option uses both a Verilog-AMS wrapper with purely digital ports and a SystemC shell for Verilog-AMS modules that are instantiated in SystemC models. The Verilog-AMS wrapper has only digital ports, so analog signals in the Verilog-AMS module instantiated within it must be converted to digital signals by connect or interface modules. The connect or interface modules are instantiated in the Verilog-AMS wrapper. The SystemC shell provides the mechanism for instantiating the Verilog-AMS wrapper in the SystemC model.



The hierarchy of shells and wrappers for instantiating a VHDL-AMS module (not illustrated here) parallels that for Verilog-AMS.

You can create the Verilog-AMS wrapper and the SystemC shell using the `xmshell` command.

The wrapper generated by the `xmshell` command is an ordinary Verilog-AMS module (but with purely digital ports), which appears in scopes and in hierarchical names. Except when the port of the imported module is analog, each port of the wrapper has the same direction as the corresponding port of the imported Verilog-AMS or VHDL-AMS module. When the port of the imported module is analog, the port of the wrapper always has the direction `inout`. This practice has the advantage of better supporting driver-receiver segregation, but the disadvantage of allowing only bidirectional connect modules to connect to the port of the wrapper.

Preparing Interface Modules

Interface modules are custom-designed modules used to establish the digital-to-analog and analog-to-digital data communication required to connect Verilog-AMS or VHDL-AMS ports to SystemC nets. Interface modules have these characteristics:

- Each interface module can be declared either as a `module` or a `connectmodule`. The latter declaration creates a true connect module capable of driver-receiver segregation. The former creates a more flexible module, but one that is incapable of driver-receiver segregation.
- Interface modules must have two ports, one declared as analog and the other as non-analog. The analog port can have any kind of scalar or vector discipline. The non-analog port can have any kind of discrete discipline.
- Interface modules can use any kind of analog to non-analog and non-analog to analog conversion logic.

If a connect module provides the characteristics you need, you can take advantage of the support that connect modules supply for driver-receiver segregation. Like interface modules, you specify connect modules on the `xmshell` command using the `-analogim` option.

You can only use connect modules, however, if the port of the imported module and the net of the SystemC model are compatible. The compatible port and net values are listed in the same row of the following table.

SystemC Net	Description	VerilogEquivalent	Verilog-AMS, VHDL-AMS port
bool, sc_bit	2-state Boolean	logic wire	scalar analog
sc_logic	4-state logic	logic wire	scalar analog
sc_bv<W>	vector of 2-state	logic [W-1 : 0]	analog [W-1 : 0]
sc_lv<W>	vector of 4-state logic	logic [W-1 : 0]	analog [W-1 : 0]
sc_int<W>	<W> bit signed integer	logic [W-1 : 0]	analog [W-1 : 0]
sc_uint<W>	<W> bit unsigned integer	logic [W-1 : 0]	analog [W-1 : 0]
char	character/8-bit signed int	logic [7:0]	analog [7 : 0]

unsigned char	character/8-bit unsignedint	logic [7:0]	analog [7:0]
short	16-bit signed integer	logic [15:0]	analog [15:0]
unsigned short	16-bit unsigned integer	logic [15:0]	analog [15:0]
sc_fixed<W,I,Q,O,N>	templated signed fixedpoint	logic [W-1 : 0]	analog [W-1 : 0]
sc_unfixed<W,I,Q,O,N>	templated unsignedfixed point	logic [W-1 : 0]	analog [W-1 : 0]
int , long	32 bit integer	logic [31:0]	analog [31:0]
unsigned int, unsigned long	32-bit unsigned integer	logic [31:0]	analog [31:0]
long long	64-bit signed integer	logic [63:0]	analog [63:0]
unsigned long long	64-bit unsigned integer	logic [63:0]	analog [63:0]
double	real data type	logic [63:0]	analog [63:0]
float	real data type	logic [63:0]	analog [63:0]

If you need to connect combinations other than those in the table, an interface module gives you more flexibility.

Example: Interface Module for Connecting SystemC Double to Verilog-AMS Electrical

This example assumes that you need to instantiate a Verilog-AMS module in a SystemC model and that the `electrical` port of the Verilog-AMS module is to be connected to a real `double` signal in the SystemC model. The lack of support for `wreal` and 64-bit logic wires in connect modules means that this example requires an interface module.

The following interface module is one possible solution. This module converts a 64-bit vector logic signal (`lv`) into an electrical signal (`ev`).

```
`include "discipline.vams"
`include "constants.vams"

module LV2EV_64 (lv, ev);
```

```
inout      [63:0] lv;
logic      [63:0] lv;
inout      ev;
electrical ev;

real      r;

always @(lv)
begin
    r = $bitstoreal (lv);
end

analog
begin
    V (ev) <+ transition (r, 1n, 1n, 1n);
end

endmodule
```

To use this interface module, you use an `xmshell` command such as the following

```
xmshell -ams -mess -import verilog -into systemc -sctype pl:double
        -analogim p1:work.LV2EV_64 lib.vlogams_child
```

This command creates a Verilog-AMS wrapper with purely digital ports called `vlogams_child_NCSCAMS` and a SystemC shell called `vlogams_child_NCSCAMS`, and uses these to connect the electrical port and the double signal.

Guidelines for Using AMS Modules in SystemC Models

The following guidelines apply when using SystemC models:

- SystemC models do not fully support automatically-inserted connect modules. The elaborator does not support using an automatically-inserted connect module when used between a Verilog-AMS or VHDL-AMS module and a digital net that runs all the way through the hierarchy into the SystemC level. The elaborator does support manually inserted connect modules in this situation, but then driver-receiver segregation does not occur and the connect modules are handled as ordinary modules. If the digital net does not reach into the SystemC level, both manually and automatically inserted connect modules are fully supported.
- SystemC models do not support `wreal` values in imported Verilog or Verilog-AMS modules. The `xmshell` program does not allow you to import Verilog or Verilog-AMS modules that use `wreal` ports. This limitation precludes connecting SystemC double values with Verilog or Verilog-AMS electrical or `wreal` values.

- SystemC models do not support passing parameters to interface modules. No means is provided in the `xmshell` program to pass parameters to interface modules.
- SystemC models offer limited compatibility checking. The `xmshell` program performs only limited checking on the compatibility of interface module ports and imported module ports. The `xmshell` program does not check data type or discipline compatibility.
- Discipline resolution results are not available to the `xmshell` program. Ports declared as `wire` in imported modules can resolve to either continuous or discrete after discipline resolution. The `xmshell` program, however, is used prior to discipline resolution and so the ultimate discipline of ports declared as `wire` is unknown. The `xmshell` program assumes that a wire is non-analog and connects the non-analog port of the interface module to it. This choice might, or might not, be correct.

Related Topic

- [The `xmshell` Command](#)

The `xmshell` Command

The `xmshell` command is used to create the Verilog-AMS wrapper and the SystemC shell.

`xmshell` Command Syntax

For importing Verilog-AMS modules:

```
xmshell_command ::=  
    xmshell{xmshell_param}-ams-importverilog-into systemc  
    lib . module {analogim_param}  
  
analogim_param ::=  
    -analogim child_port [@interface_mod_port]  
    : lib2 . interface_mod [:view]
```

For importing VHDL-AMS modules:

```
xmshell_command ::=  
    xmshell{xmshell_param}-ams -importvhdl -intosystemc  
    lib . module {analogim_param}  
  
analogim_param ::=  
    -analogim child_port [@ interface_mod_port]
```

```
: lib2 . interface_mod [:view]
```

xmshell Command Options

This section describes the options that you can use with the `xmshell` command.

<code>xmshell_param</code>	An additional <code>xmshell</code> option that you might wish to use. For a list of available options, use <code>xmshell -help</code> . For example, you might use the <code>-messages</code> or <code>-sctype</code> options.
<code>lib</code>	Library containing the Verilog-AMS or VHDL-AMS module that is to be imported into the SystemC model.
<code>module</code>	Verilog-AMS or VHDL-AMS module that is to be imported into the SystemC model.
<code>-analogim</code>	Parameter indicating that <code>lib2.interface_mod</code> is to be used as the interface module for <code>child_port</code> .
<code>child_port</code>	Port of the imported Verilog-AMS or VHDL-AMS for which the <code>interface_mod</code> module is inserted. If <code>child_port</code> is specified as <code>*</code> , the <code>interface_mod</code> module is inserted in the Verilog-AMS wrapper for each port of <code>module</code> .
<code>interface_mod_port</code>	Name of the interface module port to be bound to <code>child_port</code> . If <code>interface_mod_port</code> is not specified, the <code>xmshell</code> program uses whichever of the two ports of <code>interface_mod</code> is compatible with <code>child_port</code> : the analog port is connected to child ports declared as analog; otherwise, the discrete port is connected.
<code>lib2</code>	Library containing <code>interface_mod</code> .
<code>interface_mod</code>	The interface module to be used to translate analog signals from the Verilog-AMS or VHDL-AMS module into digital signals, or vice versa.
<code>view</code>	View of <code>interface_mod</code> to be used as the interface module.

Examples of the xmshell Command

The following examples illustrate different ways of using the `xmshell` command, in particular with the `-analogim` option.

Example: Connecting an Interface Module Default Port

The following example creates a Verilog-AMS wrapper with purely digital ports and a SystemC shell for `lib.vlogams_child`. The port `p1`, of the `vlogams_child`, is connected to the appropriate port of the `work.LV2EV_64` interface module (if `p1` is analog, it is connected to the analog port of the interface module; if `p1` is not analog, it is connected to the non-analog port of the interface module).

```
xmshell -ams -mess -import verilog -into systemc -sctype p1:double  
        -analogim p1:work.LV2EV_64 lib.vlogams_child
```

Example: Connecting a Specified Port of the Interface Module

The following example connects the appropriate port of the interface module `work.L2E` to the `p1` port of the `vlogams_child`. If `p1` is analog, the analog port of `work.L2E` is connected with `p1`; otherwise, the non-analog port of `work.L2E` is connected to `p1`. The explicit port map `p2@Px:work.L2E` connects the `p2` port of `lib.vlogams_child` to the `Px` port of interface module `work.L2E`.

```
xmshell -messages -AMS -import verilog -into systemc -sctype p1:bool  
        -sctype p2:sc_bit -analogim p1:work.L2E  
        -analogim p2@Px:work.L2E lib.vlogams_child
```

Related Topic

- [Instantiating Verilog-AMS and VHDL-AMS in SystemC](#)

Incremental Elaboration for Mixed-Signal

Multi-Snapshot Incremental Elaboration (MSIE) is a technology that enables you to partition the design into sections based on the stability of the code. Parts of the design that are stable, or require only occasional changes, are placed into one or more primary partitions, while parts of the design that undergo frequent changes are placed into a separate incremental partition. Each partition is elaborated separately into a snapshot, and the snapshots are then combined at simulation time. When a change is made, only the affected partition is re-elaborated.

Incremental elaboration is also enabled for designs with mixed-signal behavior (either AMS or DMS).

Verilog-AMS is supported in primary partitions with the following restrictions:

- It is not allowed in the top-level modules of a primary partition if the module has ports.
- It is not allowed in modules containing instances that are left unbound using the `-incrbind` option.
- SV-DC interconnect nets are not allowed in modules containing instances that are left unbound using the `-incrbind` option.
- Wires in a top-level module of a primary partition do not coerce to `electrical` or `wreal` if the module has ports.
- Wires in modules containing instances that are left unbound using the `-incrbind` option does not coerce to `electrical` or `wreal`.

Verilog-AMS is supported in incremental partitions with the following restrictions:

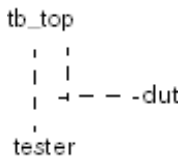
- It is not allowed in modules that bind to unbound instances in a primary partition.
- It is not allowed in modules containing instances that bind to primary partitions.
- SV-DC interconnect nets are not allowed in modules containing instances that bind to primary partitions.
- Wires in modules that bind to unbound instances in a primary partition do not coerce to `electrical` or `wreal`.
- Wires in modules containing instances that bind to primary partitions do not coerce

to `electrical` or `wreal`.

Use Models of Incremental Elaboration

Partitioning for UVM Test Modifications

Consider the following diagram where `tb_top` and `dut` are the primary partitions and `tester` is the incremental partition.

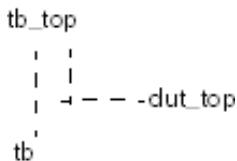


Here are how the restrictions apply for this example:

- Verilog-AMS is not allowed in `tb_top` and `tester`
- Wires in `tb_top` do not coerce to `electrical` or `wreal`
- Wires in `tester` do not coerce to `electrical` or `wreal`
- SV-DC interconnect nets are not allowed in `tb_top`

Partitioning for a Stable DUT

Consider the following diagram where `dut_top` is the primary partition and `tb_top` and `tb` are incremental partitions.

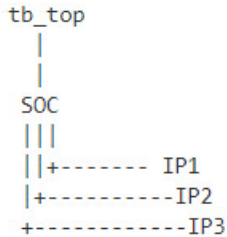


Here are how the restrictions apply for this example:

- Verilog-AMS is not allowed in `tb_top` and in the top-level of `dut_top`
- Wire in the top-level of `dut_top` does not coerce to `electrical` or `wreal`
- Wires in `tb_top` do not coerce to `electrical` or `wreal`
- SV-DC interconnect nets are not allowed in `tb_top`

Flexible Partitioning for Complex SOCs

In the following diagram, `tb_top` and `SOC` are in one primary partition, while `IP1`, `IP2`, and `IP3` are individual primary partitions.



The following restrictions apply for this example:

- Verilog-AMS is not allowed in top-levels of `IP1`, `IP2`, or `IP3`
- Wires in top-levels of `IP1`, `IP2`, or `IP3` do not coerce to `electrical` or `wreal`
- Verilog-AMS is not allowed in the modules in `SOC` that instantiate `IP1`, `IP2`, and `IP3`
- Wires in the modules in `SOC` that instantiate `IP1`, `IP2`, and `IP3`, do not coerce to `electrical` or `wreal`
- SV-DC interconnect nets is not allowed in the modules in `SOC` that instantiate `IP1`, `IP2`, and `IP3`

Unsupported Scenarios of Incremental Elaboration for Mixed-Signal

The following are the unsupported scenarios:

- The `-autospiceoomr` and `-ignore_spice_oomr` command-line options to manage Out-of-Module References (OOMR) to SPICE in digital statements
- Low-power mixed-signal verification with multiple snapshots. See the limitations listed in the [Incremental Elaboration in Low-Power Mixed-Signal Designs](#) section.
- VHDL terminal port and VHDL-AMS on primary top
- Array of instances
- Discipline resolution at partition boundary port or net

- Auto-MSIE functionality in designs containing Verilog-AMS or SPICE
- The `-spicetop` option to run a SPICE-on-Top design
- AMS Virtuoso use model

Limitations of Incremental Elaboration for Mixed-Signal

Currently, the following limitations exist in incremental elaboration:

- Inherited connections must resolve to a net in the same partition as they exist either through defaults or through overrides. This prevents discipline resolution from propagating across the partition boundary; and, also eliminates the need for an interface element across the partition boundary. This requires that the global power supply nets in the `cds_globals` module is included as another top-level module in the primary partition.
- Generating an `ieinfo` report during primary elaboration is not supported. When an `ieinfo` report is requested during the final elaboration, the list of interface elements also includes the ones from the primary partition. In addition, the hierarchical paths in the report are adjusted to the full path in the final design.
- The `-amspartinfo` option is currently not supported.
- Multiple `xrun` flows are not supported for SPICE configurations.
- OOMR connection in incremental partition referring to an object of type `electrical`, `wreal` in the primary partition or vice versa.
- OOMR connection in primary partition referring to an object of type `analog`, `coerced wreal`, or `coerced UDN` in the incremental partition.
- OOMR connection in primary partition referring to an object of type `wudn` in the incremental partition.
- OOMR connection to an analog node in the elaborated primary boundary.
- Analog OOMR connection to elaborated primary boundary or through internal `top_externs` module.
- Out of Primary References (OOPRs) from one primary partition to another.

- Connect modules at partition boundary.
- SystemVerilog interconnects at partition boundary.
- Discipline resolution of electrical nets at partition boundary.
- Verilog-AMS module with ports as a primary or incremental snapshot.
- SystemVerilog `wire` connect to VHDL `real` port at partition boundary.
- An incremental instantiate primary flow with an instance-based setting in the AMSD block, such as a config card or `ie` card to do BDR setting, is not supported in primary elaboration. Instead, you must use the relative path of the primary top.
- Programmable RNM coercion.
- Bind statement in incremental partition and bind target in a primary partition.
- SV bind statements on SPICE blocks (with `-xmcbind`) on a spice internal node.
- AMS and DMS nets cannot be declared or resolved across partition boundaries.
- SV wire connects to VHDL real port at the partition boundary.
- SPICE configurations in primary partition under multiple run mode.
- Back to Back interface elements (IE) at partition boundary.
- Binding of Verilog AMS module to primary or incremental snapshot target modules.
- RNM (wreal) coercion on specific scope (`-rnm_coerce on scopeType-scope`) is not supported in the single xrun flow.

Related Topics

- [Incremental Elaboration at SV-RNM Partition Boundary](#)
- [Incremental Elaboration in OOMR Connections](#)
- [Cloning of Replicated Modules](#)
- [Incremental Elaboration in Designs with UNL Netlists](#)

Incremental Elaboration at SV-RNM Partition Boundary

Incremental elaboration is enabled for designs containing SystemVerilog User Defined Nettype (SV-UDN) ports at the partition boundaries. You can specify UDN ports at the primary partition boundary of a design; then, the UDN ports from the different secondary partitions are connected and merged into one simulation net.

Verilog-AMS and Spice instantiations can be instantiated within partitions; however, these ports are not allowed across partition boundaries.

The following example enables IE for the SV-UDN port type and `xrun` compiles the `top.sv` file. Module, `top` is the incremental partition and `driver1` is the primary partition:

```
xrun -clean top.sv -primtop driver
//top.sv
//incremental
module top;
WUDNTYPE w;
assign w = 1.1;
driver1 d1(w);
endmodule

//primary
module driver1 (dr_1);
input dr_1;
WUDNTYPE dr_1;
Endmodule
```

The following are the port types that are supported for incremental elaboration at SV-RNM partition boundary:

- `sv wreal`. For example:

```
import cds_rnm_pkg::*;
wrealsum w;
or
wrealavg w;
```

- `nettype real` with UDR. For example:

```
function automatic real FUNC(input real drivers[]);
begin
FUNC = 1.0;
foreach (drivers[j])
```

```
begin
$display("\nFUNC : drivers[%d] = {%f} ", j, drivers[j]);
FUNC += drivers[j];
end
end
endfunction

// A nettype declaration with datatype
nettype real WUDNTYPE with FUNC;
```

- **net type struct with one real field, with UDR.** For example:

```
typedef struct {
real V;
} Rstruct;

function automatic Rstruct FUNC(input Rstruct drivers[]);
begin
FUNC = '{1.0}';
foreach (drivers[j])
begin
$display("\nFUNC : drivers[%d] = {%f} ", j, drivers[j].V);
FUNC.V += drivers[j].V;
end
end
endfunction

// A nettype declaration with datatype
nettype Rstruct WUNDTYPE with FUNC;
```

- **nettype struct with multiple fields.** For example:

```
typedef struct {
real field1;
real field2;
}T;

// user-defined resolution function Tsum
function automatic T Tsum (input T driver[]);
begin
Tsum.field1 = 0.0;
Tsum.field2 = 0.0;
foreach (driver[i])
begin
$display("driver[%d]{%f}{%f}", i, driver[i].field1, driver[i].field2);
Tsum.field1 += driver[i].field1 ;

```



```
Tsum.field2 += driver[i].field2 ;
end
$display("Tsum{%f}{%f}", Tsum.field1, Tsum.field2);
end
endfunction

// A nettype declaration with datatype and resolution function
nettype T WUDNTYPE with Tsum;
OR:
import EE_pkg::*;
EEnet w;
```

- nettype without UDR. For example:

```
nettype real WUDNTYPE;
```

The following are the restrictions of using SV-UDN ports at the partition boundaries:

- If one side of the connect is UDN and the other side cannot be UDT (struct variable).
- Wreal type in .vams connection is not supported. In such cases, you must transfer .vams to .sv; and, use SV Wreal.
- Datatype connections such as real connect to non-real, UDN connect to non-UDN are not supported at partition boundaries.
- Interconnect coercion to nettype that is already elaborated in a primary partition is not supported.
- Discipline resolution of electrical nets at partition boundary.

Related Topics

- [Auto-Partitioning Designs with SV-RNM Ports](#)
- [Coercion of User-Defined Nettype \(UDN\) across Partition Boundaries](#)

Auto-Partitioning Designs with SV-RNM Ports

You can do auto-partitioning of designs that contain SystemVerilog User Defined Nettype (SV-UDN) ports at the partition boundaries.

The auto-partitioning solution in multi-snapshot incremental elaboration (MSIE), also known as Auto-MSIE, is provided for automatically creating optimum partitions in the design. It divides an entire test case into multiple primary partitions.

To enable auto-MSIE for SV-RNM designs, use any of the following `xrun` command-line options:

- `-genpartition`: The tool creates a partition file and then you have to pass this file to `xrun` to initiate incremental elaboration

```
xrun <source-filename> -genpartition <partition-filename>
```

where:

- `source-filename` is the file where the design is specified
- `partition-filename` is the name of the partition file the tool creates. It can be an absolute or relative path

For example:

```
xrun top.sv -genpartition a.f
```

Once the tool generates the partition file, you can review and edit the file, if required. Then, you must manually specify the file to the tool to build the MSIE model, using the following command:

```
xrun top.sv -f a.f
```

By default, the tool creates a maximum of 12 primary partitions in a design. However, you can use the `-xmsienumpartition <num>` option to specify the number of partitions.

- `-automsie`: This option is a one-step methodology where the tool automatically creates a partition file and initiates incremental elaboration.

By default, the tool creates a maximum of 3 primary partitions in a design.

Additionally, you can also specify the number of primary parallel elaborations to be run using the `-primparallelelab` command-line option.

The auto-MSIE functionality is not supported in the following scenarios:

- Design containing Verilog-AMS or SPICE
- Coercion of `wreal` nets at partition boundary
- Blocked based discipline setting (Multiple `ie` cards)

Related Topics

- [Incremental Elaboration for Mixed-Signal](#)
- [Incremental Elaboration at SV-RNM Partition Boundary](#)

Coercion of User-Defined Nettype (UDN) across Partition Boundaries

For SV-UDN designs, RNM coercion is allowed at partition boundaries with incremental elaboration; and, coercion can be propagated across the partitions.

The following is an example of the incremental instantiating primary partition. Here, `top.w` and `top.mid.w` are coerced across partition boundaries as nettype `wrealsum`.

```
xrun -clean top.sv -prmtop prim

//top.sv file
//incremental
module top;
wire w; //"w" is coerce to wrealsum
mid m1(w);
endmodule // top

module mid(w);
inout wire w; //"w" is coerced to wrealsum
prim p1(w); //primary partition
endmodule // mid

//primary top
module prim(w);
inout wire w; //"w" is coerced to wrealsum
leaf l1(w);
endmodule // prim

module leaf(w);
import cds_rnm_pkg::*;
inout wrealsum w;
assign ww=1.1;
endmodule
```

Limitations

- Verilog-AMS wreal cannot be coerced across partition boundaries.
- SV wire at partition boundary that is coerced from Verilog-AMS wreal cannot coerce across the partition.
- Coercion of SV interconnects to nettypes across partition boundaries is not supported. You can use wires in place of SV interconnects.
- Connecting SV Wreal to SV real variable and SV-UDN to SV-UDT are not supported.
- Nettype defined within modules cannot be coerced across partition boundaries.
- Binding instance through SV bind across partition boundaries is not supported.

Related Topic

- [Incremental Elaboration for Mixed-Signal](#)

Incremental Elaboration in Designs with UNL Netlists

Incremental Elaboration can be enabled in designs with UNL netlist in the primary partition and UVM or SV testbench in the incremental partition. To enable incremental elaboration, you must use the `-relax_stiavm_multirun` command-line option.

You must use the multi-step invocation mode, to create snapshots and run the simulation.

Single-step invocation mode is currently not supported in these designs.

To enable incremental elaboration in the multi-step mode, perform the following steps:

1. Create the netlist and generate the ADE/UNL command-line options to build the primary partition.
2. Build the primary snapshot.

Ensure that you use the `-relax_stiavm_multirun` command-line option at this step. Also, the UNL top must be defined in the primary top.

3. Build the incremental snapshot.
4. Run the simulation.

For advanced testbench reuse, you must build Virtuoso IP1 and IP2 as the primary snapshots and the top-level testbench as the incremental snapshot.

Related Topics

- [Incremental Elaboration at SV-RNM Partition Boundary](#)
- [Incremental Elaboration in OOMR Connections](#)

Incremental Elaboration in OOMR Connections

You can enable incremental elaboration for designs with OOMR in the incremental partition referring to resolved `analog`, `coerced wreal`, and `coerced UDN` in primary partitions. Incremental elaboration is also supported in an OOMR connection with a bidirectional `tran` primitive that is referring to an object in a primary partition.

Currently, incremental elaboration in OOMR connections is supported only in the single-run flow.

Examples

The following is an example of incremental elaboration in a design with SV `wreal` in the primary partition, where `p1.pb0.clk` is coerced to `wreal` in the primary partition.

```
xrun prim.sv incr.sv primary_bottom.vams -prmtop primary -rnm_tech
//incremental partition
module incremental();
wire status;
assign status=p1.pb0.clk;
primary p1();
endmodule

//primary partition
import cds_rnm_pkg::*;
module primary();
wire status;
primary_middle pb0(status);
endmodule

module primary_middle(inout clk);
primary_bottom pb2(clk);
endmodule

module primary_bottom(inout bclk);
wrealsum bclk;
endmodule
```

The following is an example of incremental elaboration in a bidirectional tran gate primitive connection in a primary partition.

```
xrun top.sv prim.sv xprim.vams -top top -prmtop prim ....
//top.sv
module top;
tran t1(q, prim.x1.aa);
prim prim();
endmodule

//prim.sv
module prim;
wire w, x, y, z;
xprim x1(w, x, y, z);
endmodule

//xprim.vams
module xprim(w,x,y,z);
output w,x,y,z;
wire aa;
endmodule
```

Related Topics

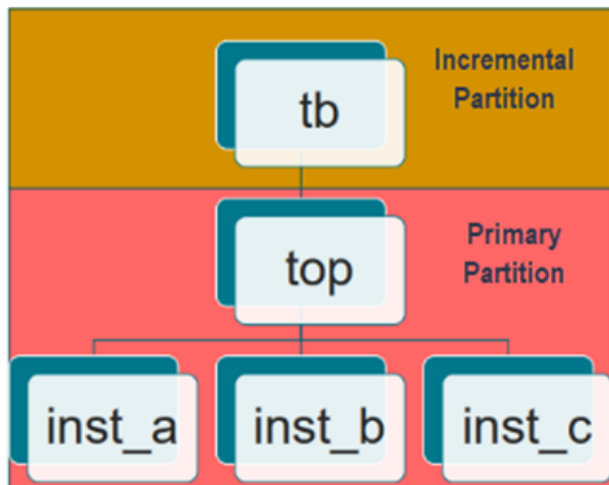
- [Incremental Elaboration for Mixed-Signal](#)
- [Cloning of Replicated Modules](#)

Incremental Elaboration in Low-Power Mixed-Signal Designs

You can enable incremental elaboration in low-power mixed-signal designs with SV-RNM datatypes. For example:

```
xrun tb.sv top.sv -prntop top -lps_ams_connect_supply -lps_ams_sim -lps_1801 dut.upf
```

Currently, SVRNM power-intent is supported only in the primary partition as illustrated in the following figure. This means that the modules for which power intent is defined in the UPF file should always be in the primary partition and there must always be a top module before the module that has power-intent specified in UPF file.



Power intent inside primary can include the followings low-power mixed-signal features:

- SV-wreal power supply net
- SV-UDN power supply net
- Power smart interface elements
- LP VPI tasks driving UPF supply net
- Mixed-signal driving UPF supply net

The following are the limitations of enabling incremental elaboration in low-power mixed-signal designs:

- Limitations from dual snapshot simulation (DSS) + Real Number Modeling (RNM)
- Limitations from monolithic Low-power + RNM
- Any SV-RNM power connection across the partition.
- Power intent in incremental (-lps_idss)

The following is an example of a low-power mixed-signal design with SV-RNM datatypes. The module, `top_wrapper` has power intent for this module defined in the UPF file. And, `top_wrapper` module is instantiated in the module `tb`. Module `top_wrapper` is specified as the primary partition and module `tb` must be in the incremental partition.

```
#####tb.sv#####
module tb ();
    top_wrapper top_wrapper ();
endmodule
```

```
module top_wrapper ();
    top top ();
endmodule
```

```
#####top.sv#####
`timescale 1ns / 100ps
import EE_pkg::*;
module top ();

    initial begin
        #200
        $supply_on("VDD", 1.0);
        $supply_on("VSS", 0.0);
        #200
        $supply_off("VDD");
        $supply_off("VSS");
    end

    drvr idrvr ();
    load iload ();
endmodule

module drvr (vdd, vss);
    output vdd, vss;
    EEnet vdd, vss;
    real v1, v2, r;
```



```

initial begin
    v1 = 0.0;
    v2 = 0.0;
    r = 10000000;

    #100 v1 = 1.8;
        r = 200;
    #200 v1 = 1.5;
    #200 v1 = 0.0;
        r = 10000000;
    #100 $finish();
end

assign vdd = '{V:v1, I:`wrealZState, R:r};
assign vss = '{V:v2, I:`wrealZState, R:r};
endmodule

module load (vdd, vss);
    input vdd, vss;
    EEnet vdd, vss;
    EEnet evdd, evss;
    assign evdd = vdd;
    assign evss = vss;
endmodule

-----wrapper.upf-----
upf_version 2.0
set_design_top tb/top_wrapper -testbench
set_design_top top
set_scope
create_supply_net VDD
create_supply_net VSS
create_supply_set SSD -function {power VDD} -function {ground VSS}
create_power_domain PDD -supply {primary SSD} -include_scope
connect_supply_net VDD -ports {idrvr/vdd}
connect_supply_net VSS -ports {idrvr/vss}
connect_supply_net VDD -ports {iload/vdd}
connect_supply_net VSS -ports {iload/vss}

-----command line-----

xrun -clean $CDS_INST_DIR/tools/affirma_ams/etc/lpms/UPF_EEnet.svams tb.sv top.sv -
access rwc -timescale 1ns/100ps -auto_config_svams_ie -lps_1801 wrapper.upf -
lps_ams_connect_supply -lps_ams_sim -lps_ams_supply_ie -lps_dut_top top_wrapper/top -
prmtop top_wrapper -dms_rnm_upf_dss -rnm_dmsie_msie -svams_2019

```

Related Topic

- [Incremental Elaboration for Mixed-Signal](#)

Cloning of Replicated Modules

In the monolithic elaboration flow of Xcelium, the entire design hierarchy is elaborated. This leads to repetitive work for designs that have a high replication factor.

The cloning enhancement improves elaboration performance in designs where a module in the design hierarchy is instantiated many times. Cloning the replicated module ensures that only one instance of the replicated module is elaborated in the form of a snapshot, and the same snapshot is then cloned to fill in for other instances. Therefore, the tool avoids elaborating repetitive instances of the same module.

In mixed-signal designs, you can clone a replicated module using the following two command-line options:

- `-replicated_top <arg>`: Allows you to specify the module that has to be cloned. You must specify the module name using the `<arg>` parameter.

You can also specify multiple modules that are anticipated to have high replication as shown below:

```
-REPLICATED_TOP mod_a \  
-REPLICATED_TOP mod_b \  
-REPLICATED_TOP mod_c \  
...
```

Internally, Xcelium would skip elaborating repetitive instances of these specified modules and clone the first elaborated snapshot of each such module whenever further instantiations are encountered.

- `-autoreplicated_top`: Specifies the tool to automatically clone a replicated module. Then, the tool elaborates the replicated module only once and re-uses the elaborated snapshot by cloning and stitching these clones towards the end of the elaboration process. If two or more modules are instantiated multiple times in a design, the tool chooses to clone the highest instantiated module.

Currently, the `-autoreplicated_top` option is supported only for module instances that are in the range, 2 to 64.

The following are some limitations with the cloning of modules:

- Replication by an array of instances (AOI)
- Designs that have enabled programmable coercion off/on
- Instance-based *ie* card setting or cell setting that covers replicated top
- SPICE and VAMS designs

Examples

The following is an example of a manual instance replication using the `-autoreplicated_top` option, where the module `driver1` has been instantiated three times.

```
xrun top.sv -rnm_tech -autoreplicated_top module top;

import cds_rnm_pkg::*;
wrealsum w;
driver1 d1(w);
driver1 d2(w);
driver1 d3(w);
endmodule

// primary
module driver1 (dr_1);
import cds_rnm_pkg::*;
  inout dr_1;
  wire dr_1;
endmodule
```

The following is an example of a for-loop construct replication using the `-autoreplicated_top` option, where the module `items` have been instantiated multiple times.

```
xrun top.sv -rnm_tech -autoreplicated_top

import cds_rnm_pkg::*;
module top();
  wrealsum a;
  genvar i;
  for(i=0; i<31;i++)
    mid x1_0(a);
endmodule

module mid(inout clk);
  wire clk;
endmodule
```

Related Topics

- [Incremental Elaboration for Mixed-Signal](#)
- [Incremental Elaboration at SV-RNM Partition Boundary](#)

Appendix A: Updating Legacy Libraries and Netlists

Before using existing libraries with the AMS Designer simulator, you may have to make modifications to the libraries. The following are some of the modules and libraries are that you can modify:

- **Verilog-A Modules:** The Verilog[®]-A language is a subset of Verilog-AMS, but some of the language elements in that subset have changed since the release of Verilog-A. As a result, you might need to revise your Verilog-A modules before using them as Verilog-AMS modules. For more information, see [Updating Verilog-A Modules](#)

In addition, some Verilog-A modules can be made more efficient by rewriting them to take advantage of the digital and mixed-signal aspects of Verilog-AMS. In these cases, you might want to generate an alternate cellview for use with the AMS Designer simulator. For more information, see [Mixed-Signal Aspects of Verilog-AMS](#).

- **Libraries of Analog Masters:** The AMS Designer simulator uses analog primitive tables to accelerate the processing of Spectre/SPICE model files. Each model card has a unique name, referred to as the analog master, which allows it to be accessed by Verilog-AMS. Before you can use model files in your designs you must create analog primitive table files for them. See [Analog Primitives and Subcircuits](#).
- **Verilog Modules:** A module written for the purely digital Verilog language can often be used unchanged in the AMS Designer simulator. However, it might be necessary to make some minor changes, such as escaping or modifying keywords, to make the module legal for both Verilog and Verilog-AMS. If it is not possible to modify a Verilog module to make it compliant with both languages, you can use the unmodified file by compiling it without using the `-AMS` option for the `xrun/xmvlog` command.

For example, you might have a Verilog module that uses `branch` as a variable. That is legal in Verilog but illegal in Verilog-AMS (which recognizes `branch` as a keyword). As a result, you must compile the module without using the `-AMS` option so that the module is compiled as Verilog, not as Verilog-AMS.

- **VHDL Blocks:** With the Spectre AMS Designer Simulator, you can use VHDL textual data directly, without importing models. Any VHDL block that runs with the NC-VHDL simulator runs with the AMS Designer simulator too.

- **Legacy Netlists:** Netlists used for analog-only simulators, such as the Spectre® circuit simulator, serve a number of purposes, including instantiating components, setting initial conditions, defining models, and specifying analyses. In the Spectre®AMS Designer simulator, instantiation and model specification are separated from the simulator controls and some of the analog controls are separated from the rest of the simulation controls. As a result, the primary way of controlling the analog solver is to define an analog simulation control file. The controls you can use in the analog simulation control file differ slightly from those in netlists, so you might need to rewrite legacy netlists to use the controls described in [Controls for the Analog Solvers](#).

The AMS Designer simulator does not support everything used in existing netlists. For information about unsupported features, see the [Spectre AMS Simulator Known Problems and Solutions](#).

- **Existing Designs:** Designs entered using Virtuoso Schematic Editor in flows such as the Cadence® analog design environment flow are automatically translated to Verilog-AMS netlists. Issues including some of those mentioned above must first be addressed before the AMS netlister can properly work. For guidance about complying with the AMS design guidelines, see the "Designing for Virtuoso AMS Compliance" chapter, of *Virtuoso AMS Environment User Guide*.

Appendix B: Tcl-Based Debugging

This appendix describes the Tcl-based simulator commands you can use to debug your design. Tcl is an object-oriented language, which means that you supply the action to be performed on the object as a modifier to the command.

The command format is

```
xcelium> Tcl_command [ -modifiers ] [-options] [arguments]
```

For information on the Tcl commands you can use with the Spectre[®] AMS Designer simulator, see the following table. Some commands are detailed here and have links to the respective topics. For commands that do not have a link, and for other commands you can use only with purely digital designs, see ["Using the Tcl Command-Line Interface"](#) in the *Xcelium Simulator Tcl Command Reference*.

Tcl Command	Description
<code>alias</code>	Defines aliases that you can use as command short-cuts.
<code>analog</code>	Controls the analog solver during mixed-signal simulation using AMS simulator with Spectre/APS solver.
<code>attribute</code>	Enables VHDL function-valued attributes for specified signals so that they can be accessed from the Tcl interface with the <code>value</code> command. The <code>attribute</code> command is enabled only for purely digital designs.
<code>call</code>	Lets you call a user-defined C-interface function or a Verilog user-defined PLI system task or function from the command line.
<code>coverage</code>	For purely digital designs, controls the dumping of code coverage data.
<code>database</code>	Lets you control an SHM or VCD database. An SHM database can hold both analog and digital databases. VCD databases do not support both analog and digital databases in a single database.
<code>deposit</code>	Lets you set the value of an object.
<code>describe</code>	Displays information about the specified simulation object, including its declaration.

<code>drivers</code>	Displays a list of all contributors to the value of the specified objects. When the simulator is stopped in a resolution function (<code>at_resfunc</code>), the TCL <code>drivers</code> command can be invoked without arguments. See at_resfunc .
<code>exit</code>	Terminates simulation and returns control to the operating system.
<code>finish</code>	Causes the simulator to exit and returns control to the operating system.
<code>fmibkpt</code>	Performs operations on breakpoints that are coded into C models using the <code>fmiBreakpoint</code> call.
<code>force</code>	Sets a specified object to a given value and forces it to retain that value until it is released with a release command or until another force is placed on it.
<code>help</code>	Displays information about simulator (<code>xmsim</code>) commands and options and predefined variable names and values.
<code>history</code>	Lets you re-execute commands without having to retype them.
<code>input</code>	Queues the commands in a file so that the simulator executes them when it issues its first prompt. This command is enabled only for purely digital designs.
<code>memory</code>	Loads VHDL memory from a memory file or dumps VHDL memory to a memory file. This command is disabled while the analog solver is active.
<code>omi</code>	Lets you display information about model managers and instances controlled by model managers. Also lets you pass OMI model manager run-time commands to model managers that support this capability.
<code>probe</code>	Lets you control the values being saved to a database.
<code>process</code>	Displays information about the processes that are currently executing or that are scheduled to execute at the current time. This command is disabled while the analog solver is active.
<code>release</code>	Releases any force set on the specified objects.

<code>reset</code>	Resets the currently loaded model to its original state at time zero. This command is enabled only for purely digital designs.
<code>restart</code>	Replaces the currently simulating snapshot with another snapshot of the same elaborated design. You cannot use <code>restart</code> after the current AMS simulation is started.
<code>run</code>	Starts simulation or resumes a previously halted simulation.
<code>save</code>	Creates a snapshot of the current simulation state.
<code>scope</code>	Lets you set the current debug scope, describe items declared within a scope, display the drivers of objects declared within a scope, list instances of auto-inserted connect modules within a scope, list resolved disciplines of all nets within a scope, print the source code, or part of the source code, for a scope, and display scope information.
<code>source</code>	Lets you execute a file containing simulator commands.
<code>stack</code>	Lets you view or set the current stack frame. This command is enabled only for purely digital designs.
<code>status</code>	Displays memory and CPU usage statistics and shows the current simulation time.
<code>stop</code>	Creates or operates on a breakpoint.
<code>strobe</code>	Writes out the values of objects under the control of specified conditions, changes in signal values, or at specified time intervals. This command can be used only for digital objects.
<code>task</code>	Lets you schedule Verilog-AMS tasks for execution. This command is disabled while the analog solver is active.
<code>time</code>	Displays the current simulation time scaled to the specified unit.
<code>value</code>	Prints the current value of the specified objects using the last format specifier preceding the object name argument. When the simulator is stopped in a resolution function (<code>at_resfunc</code>), the TCL <code>value</code> command can be invoked without arguments. See at_resfunc.
<code>version</code>	Displays the version number of <code>xmsim</code> .

<code>where</code>	Displays the current location of the simulation. When the simulator is stopped in a resolution function (<code>at_resfunc</code>), the TCL <code>where</code> command can be invoked without arguments. See at_resfunc.

Related Topics

- [Xcelium Simulator Tcl Commands](#)
- [Tcl Basics](#)
- [Extensions to Tcl](#)
- [Specifying Unnamed Branch Objects in Tcl Commands](#)

analog

The Tcl `analog` command controls the analog solver during a mixed-signal simulation.

analog Command Syntax

```
analog
  -iabstol <value>
  -off
  -on
  -reltol <value>
  -show
  -stop <time_spec>
  -tcl
  -vabstol <value>
```

analog Command Options

This section describes the options that you can use with the Tcl `analog` command.

<code>-ahdllint_report</code>	Prints the dynamic linter report.
<code>-iabstol <value></code>	Changes the absolute tolerance for current to the specified <i>value</i> in the next analog time step.
<code>-off</code>	Skips the analog simulation starting from this digital timepoint.

<code>-on</code>	Starts the analog simulation starting from this digital timepoint.
<code>-reltol <value></code>	Changes the relative tolerance of the analog engine to the specified <i>value</i> in the next analog time step.
<code>-show</code>	Prints the tolerance values that will be set for the next analog time step using the <code>analog</code> command, if any.
<code>-stop <time_spec></code>	Changes the analog stop time to the specified <i>time_spec</i> . The value of the time specification is numeric and must be greater than the current analog time.
<code>-tcl</code>	Switches from Xcelium interactive Tcl to the Spectre FX interactive Tcl system.
<code>-vabstol <value></code>	Changes the absolute tolerance for voltage to the specified <i>value</i> in the next analog time step.

Related Topic

- [How to Use `save`, `restart`, and `analog`](#)

call

The Tcl `call` command invokes a user-defined C-interface function, or a Verilog user-defined VPI system task or function, or a SystemC function from the command line.

call Command Syntax

```
call [-systf | -predefined] task_or_function_name [arg1 [arg2 ...]]
```

call Command Options

This section describes the options that you can use with the Tcl `call` command.

<code>-systf</code> <code>[task_or_function_name]</code>	Looks for the specified task or function name only in the table of user-defined PLI system tasks and functions. This option is available because the <code>call</code> command is also used to invoke functions from the VHDL C-interface, and there may be a user-defined C-interface function with the same name as a PLI system task or function. The <code>-systf</code> option causes the lookup in the C-interface task list to be skipped. This option must appear before the task or function name on the command line. You cannot use this option with the <code>-predefined</code> option. The command <code>call -systf</code> with no task or function, the name argument displays a list of all registered user-defined system tasks and functions.
<code>-predefined</code> <code>[function_name]</code>	Looks for the specified task or function name only in the table of predefined CFC library functions. You cannot use the <code>-predefined</code> option when calling a user-defined system task or function. This option must appear before the CFC function name on the command line. You cannot use this option with the <code>-systf</code> option. The command <code>call -predefined</code> with no function, the name argument displays a list of all predefined C function names.

call Command Arguments

Arguments to the system task or function can be either literals or names.

Literals can be:

- Integers

```
xcelium> call mytask 5
xcelium> call mytask 5 7
```

- Reals

```
xcelium> call mytask 3.4
xcelium> call mytask 22.928E+10
```

- Strings

Strings must be enclosed in double quotes. Enclose strings in curly braces or use the backslash character to escape quotes, spaces, and other characters that have special meaning to Tcl. For example:

```
xcelium> call mytask {"hello world"}
xcelium> call mytask \"hello\ world\"
```

- Verilog literals, such as `8'h1f`

Names can be full or relative path names of instances or objects. Relative pathnames are relative to the current debug scope (set by the `scope` command). Object names can include a bit select or part select. For example:

```
xcelium> call mytask top.u1
xcelium> call mytask top.u1.reg[3:5]
```

Expressions that include operators or function calls are not allowed. For example, the following two commands result in an error:

```
xcelium> call \mytask a+b
xcelium> call \mytask {func a}
```

However, literals can be created using Tcl's `expr` command. For example, if the desired argument is the expression `(a+b)`, use the following:

```
xcelium> call \mytask [expr #a + #b]
```

The result of the expression `(a+b)` is substituted on the command line and then treated by the `call` command as a literal.

The `expr` command cannot evaluate calls to Verilog functions.

If you are calling a user-defined system function, the result of the `call` command is the return value from the system function. Therefore, user-defined system functions can be used to generate literals for other commands. For example:

```
xcelium> call task [call func arg1 ...]
xcelium> force a = [call func arg1 ...]
```

call Command Examples

The following Verilog module contains a call to a user-defined system task and to a system function. The task and function can also be started from the command line.

```
module test();

initial
begin
    $hello_task();
    $hello_task($hello_func());
end
endmodule
```

The following command starts the `$hello_task` system task:

```
xcelium> call \hello_task
```

This task can also be started with any of the following:

```
xcelium> call hello_task  
xcelium> call {$hello_task}  
xcelium> call {hello_task}
```

The `$hello_func` function can be started with any of the following commands:

```
xcelium> call \hello_func  
xcelium> call hello_func  
xcelium> call {$hello_func}  
xcelium> call {hello_func}
```

In the following command, the `call` command calls the `$hello_task` system task with a call to the system function `$hello_func` as an argument.

```
xcelium> call hello_task [call hello_func]
```

The following command displays a list of all registered user-defined system tasks and functions.

```
xcelium> call -sysf
```

deposit

Sets the value of an object. Behaviors that are sensitive to value changes on the object run when the simulation resumes, just as if the value change was caused by the Verilog or VHDL code.

In this release, you cannot set the value of an analog object. In addition, you cannot use the `deposit` command to set the value of digital objects while the analog solver is active.

The `deposit` command without a delay is similar to a force in that the specified value takes effect and propagates immediately. However, it differs from a force in that future transaction on the signal are not blocked.

You can specify that the deposit is to take effect at a time in the future (`-after -absolute`) or after some amount of time has passed (`-after -relative`). In VHDL, a deposit with a delay is different from Verilog in that it creates a transaction on a driver, much the same as a VHDL signal assignment statement. Use the `-inertial` or `-transport` option to deposit the value after an inertial delay or after a transport delay, respectively.

For VHDL, you can deposit to ports, signals, and variables if no delay is specified. If a delay is specified, you cannot deposit to variables or to signals with multiple sources.

For Verilog, you can deposit to ports, signals (wires and registers), and variables.

If the object is a memory or a range of memory elements, the specified value is deposited into each element of the memory or into each element in the specified range.

If the object is currently forced, the specified value appears on the object after the force is released, unless the release value is overwritten by another assignment in the meantime.

If the object is a register that is currently forced or assigned, the `deposit` command has no effect.

The value assigned to the object must be a literal. The literal can be generated with Tcl value substitution or command substitution. (See the "Verilog Value Substitution" and "Command Substitution" sections, in the "Basics of Tcl" appendix of *Cadence Verilog Simulation User Guide* for details on Tcl substitution.)

For VHDL, the value specified with the `deposit` command must match the type and subtype constraints of the VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported. Records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

The object to which the value is to be deposited must have read/write access. An error is generated if the object does not have this access. See [Access to Digital Simulation Objects](#).

For an SV-UDN net, you can deposit value to a scalar net, whole array, or bit-select and part-select of arrays. And, you can use order-based or name-based approach to deposit a value. For the name-based approach, the filed order can be different from that defined in the SV-UDN net. You can also assign `real'z` or `real'x` values to SV-UDN.

Additionally, you can use the `xm_deposit` system task to assign value on an SV-UDN net. The syntax is: `xm_deposit ("<name>", "<value>")`. The value string should not contain any space; and, for the scalar UDN or bit-selected UDN array, the `value` string should not contain `{}`. For example:

```
$xm_deposit("dut.ve[1:2]", "{V:1.7,R:10,I:real'z}{1.7,real'x,20}");
```

The `deposit` command is supported on user-defined nettypes (UDNs).

deposit Command Syntax

```
deposit object_name [=] value
    [-after time_spec {-relative | -absolute}]
    [-inertial]
    [-transport]
    [-generic]
```

deposit Command Options

This section describes the options that you can use with the Tcl `deposit` command.

<code>-after <time_spec> [<value> -after <time_spec>...]</code>	Causes the assignment to occur at a time in the future rather than immediately. The time specified using the <i>time_spec</i> argument can be relative or absolute. Relative is the default. If you do not specify a time, the assignment happens immediately before simulation resumes. If the specified time is the current simulation time, the assignment occurs after simulation resumes but before time advances.
<code>-relative</code>	Causes the assignment to occur after the amount of time specified in <i>time_spec</i> has passed. This is the default.
<code>-absolute</code>	Causes the assignment to occur at the simulation time specified in <i>time_spec</i> .
<code>-inertial</code>	Deposits the value after an inertial delay.
<code>-transport</code>	Deposits the value after a transport delay.
<code>-generic</code>	Deposits generic value. This operation might lead to violation of globally static bounds.

deposit Command Example

Digital Verilog-AMS examples:

The following command assigns the value `8'h1F` to `r[0:7]`. No time for this assignment is specified, so the assignment occurs immediately. The equal sign is optional.

```
xcelium> deposit r[0:7] = 8'h1F
```

The following command assigns `25` to `r[8:15]` after simulation resumes and 1 time unit has elapsed.

```
xcelium> deposit r[8:15] = 25 -after 1
```

The following command assigns `25` to `r[8:15]` at simulation time 1 ns.

```
xcelium> deposit r[8:15] = 25 -after 1 ns -absolute
```

The following command sets the value of `x` to the current value of `w`. The assignment occurs at

simulation time 10 ns.

```
xcelium> deposit x = #w -after 10 ns -absolute
```

The following command uses both command and value substitution. The object `y` is set to the value returned by the Tcl `expr` command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of `r`.

```
xcelium> deposit y = [expr #r[0] & ~#r[1]]
```

The following command shows the error message that is displayed if you run in regression mode and then try to deposit a value to an object that does not have read/write access.

```
xcelium> deposit clrb 1
xmsim: *E,RWACRQ: Object does not have read/write access:
        hardrive.h1.clrb.
```

VHDL examples:

The following command deposits the value 1 to object: `t_nickel_out` (`std_logic`). The equal sign is optional.

```
xcelium> deposit :t_nickel_out = '1'
```

The following command deposits the value 1 to object: `top:DISPENSE_tempsig` (`std_logic`).

```
xcelium> deposit :top:DISPENSE_tempsig '1'
```

The following command deposits the value 0 to object: `t_dimes` (`std_logic_vector`) after 10 ns has elapsed.

```
xcelium> deposit -after 10 ns -relative :t_DIMES {"00000000"}
```

The following command deposits the value TRUE to object `stoppit` (`boolean`).

```
xcelium> deposit stoppit true
```

The following command deposits the value 10 to object: `count` (`integer`).

```
xcelium> deposit :count 10
```

SV-UDN examples:

The following example illustrates the SV-UDN structure and how to apply the `deposit` command to a scalar net with name-based value:

```
typedef struct {
```

```
real V;
real I;
real R;
} EEstruct;

Time: 0 FS: tb.ve = ('{V:0, I:0, R:1e+06}, '{V:0, I:0, R:1e+06}, '{V:0, I:0, R:1e+06})
Time: 0 FS: tb.e = '{V:0, I:0, R:1e+06}

xcelium> deposit tb.e {V:1.2, I:real'z, R:500};
Time: 40 NS: tb.e = '{V:1.2, I:`wrealZState, R:500}
```

The following example illustrates how the `deposit` command applies value on bit-select of an array with order-based value. Values 1.2, real'z, and 50 are applied on real variable V, I, and R, respectively in the order in which it is defined/declared in the structure:

```
# Bit-Selected : Order-Based Value
xcelium> deposit tb.ve[1] {1.2, real'z, 50}

Time: 40 NS: tb.ve = ('{V:0, I:0, R:1e+06}, '{V:1.2, I:`wrealZState, R:50}, '{V:0, I:0, R:1e+06})
```

The following example applies the `deposit` command on part-select of an array with order-based value:

```
xcelium> deposit tb.ve[2:1] {{V:1.0, R:10, I:real'z} {1.5, real'x, 20}}

Time: 60 NS: tb.ve = ('{V:1, I:`wrealZState, R:10}, '{V:1.2, I:`wrealZState, R:50}, '{V:0, I:0, R:1e+06})
Time: 60 NS: tb.ve = ('{V:1, I:`wrealZState, R:10}, '{V:1.5, I:`wrealXState, R:20}, '{V:0, I:0, R:1e+06})
```

Related Topics

- [deposit](#)
- [User-Defined Nettype and Resolution Function](#)

describe

Displays information about the specified simulation object, including its declaration.

- For objects without read access, the output of the `describe` command does not include the object's value.
- For objects that have read access but no write access, the string `(-W)` is included in the output.

- For objects with neither read nor write access, the string `(-RW)` is included in the output.

The `describe` command is supported on user-defined nettypes. For more information on user-defined nettypes, refer to [User-Defined Nettype and Resolution Function](#).

describe Command Syntax

```
describe simulation_object...
```

describe Command Options

None.

describe Command Examples

The section includes the examples of how the `describe` command displays information about simulation objects.

- [Verilog-AMS Examples](#)
- [VHDL Examples](#)

Verilog-AMS examples:

The following command displays information about the Verilog-AMS analog net `n1`. The resolved discipline is in parentheses. The value is the potential of the net.

```
xcelium> describe n1
n1.....wire (electrical) = 2.99227
```

The following command displays information about the Verilog-AMS digital net `p1`, which is bound to an input port.

```
xcelium> describe top.I3.clkSig
top.I3.clkSig...input (wire/tri) = St0
```

Both named and unnamed branches are described as `branch`, where the value is the potential of the branch. For example,

```
xcelium> describe top.I4.compSig_ground
top.I4.compSig_ground...branch(compSig) = 0
```

The value is the potential of the branch. To see the flow through the branch, use the `value -`

`flow` command instead.

The following command displays information about the Verilog object `data`.

```
xcelium> describe data
data.....register [3:0] = 4'h0
```

The following command displays information about two Verilog objects: `data` and `q`.

```
xcelium> describe data q
data.....register [3:0] = 4'h0
q.....wire [3:0]
  q[3] (wire/tri) = StX
  q[2] (wire/tri) = StX
  q[1] (wire/tri) = StX
  q[0] (wire/tri) = StX
```

The following command displays information about the object `alu_16`.

```
xcelium> describe alu_16
alu_16.....top-level module
```

The following command displays information about the object `u1`.

```
xcelium> describe u1
u1.....instance of module arith
```

The following commands display information about the mixed bus `w`.

```
xcelium> describe w
w.....wire (mixed bus) = Inf
xcelium> describe w[0]
w.....wire (electrical)
xcelium> describe w[1]
w.....wire [0:2]
w[1] (wire/tri) = StX
xcelium> describe w[2]
w.....wire [0:2]
w[2] (wire/tri) = StX
```

The following command displays information about the connect module `mya2d`.

```
xcelium> describe
mya2dmya2d.....instance of connect module a2d
```

The following command shows the output of the `describe` command for an object that does not have read or write access. The output of the command does not include the object's value, but the string `(-RW)`.

```
xcelium> describe d
d.....input [3:0]
  d[3] (-RW)
  d[2] (-RW)
  d[1] (-RW)
  d[0] (-RW)
```

VHDL examples:

```
xcelium> describe t_NICKEL_IN
t_NICKEL_IN...signal : std_logic = '0'
xcelium> describe t_NICKEL_IN t_CANS
t_NICKEL_IN...signal : std_logic = '0'
t_CANS.....signal : std_logic_vector(7 downto 0) = "11111111"
xcelium> describe gen_dimes
gen_dimes...process statement
xcelium> describe :top
top.....component instantiation
```

Related Topics

- [Specifying Unnamed Branch Objects](#)
- [Access to Digital Simulation Objects](#)

drivers

Displays a list of all contributors to the value of the specified digital objects.

You cannot list drivers for analog nets, analog variables, or branches.

You can use the `scope -drivers [ scope_name ]` command to display the drivers of each digital object that is declared within a specified scope. See [scope](#).

For Verilog, the `drivers` command cannot find the drivers of a wire or register unless the object has read and connectivity access. However, even if you have specified access to an object, its drivers might have been collapsed, combined, or optimized away. In this case, the output of the command might indicate that the object has no drivers. For details on specifying access to simulation objects, see [Access to Digital Simulation Objects](#).

When the simulator is stopped in a resolution function (`at_resfunc`), the TCL `drivers` command can be invoked without arguments. See [at_resfunc](#).

The `drivers` command is supported on user-defined nettypes. For more information on user-defined nettypes, refer to [User-Defined Nettype and Resolution Function](#).

drivers Command Syntax

```
drivers
  [-show] object_name...
  [-effective]
  [-future]
  [-novalue]
  [-verbose]
```

drivers Command Options

This section describes the options that you can use with the Tcl `drivers` command.

<code>-show object_name [object_name ...]</code>	Displays a list of all contributors to the value of the specified object(s). You must specify at least one object. The <code>-show</code> modifier is optional. The following two commands are equivalent: <pre>xcelium> drivers -show f af xcelium> drivers f af</pre>
<code>-effective</code>	Displays contributions to the effective value of the signal. By default, the <code>drivers</code> command displays contributions to the driving value. Only VHDL inout and linkage ports can have different driving and effective values.
<code>-future</code>	Displays the transactions that are scheduled on each driver.
<code>-novalue</code>	Suppresses the display of the current value of each driver.
<code>-verbose</code>	Displays all of the processes (signal assignment statements), resolution functions, and type conversion functions that contribute to the value of the specified signal. If you do not include the <code>-verbose</code> option, then resolution and type conversion function information is omitted from the output. This option affects VHDL signals only.

drivers Command Examples

The following examples illustrate use cases for the `drivers` command:

- [Output for Digital Verilog-AMS Signals](#)
- [Output for VHDL Signals](#)

Output for Digital Verilog-AMS Signals

The following command lists the drivers of a signal called `f`.

```
xcelium> drivers f
f.....wire (wire/tri) = StX
    StX <- (board.counter) assign fifteen = value[0] & value[1] &
        value[2] & value[3]
```

The following command lists the drivers of two signals called `f` and `af`.

```
xcelium> drivers f af
f.....wire (wire/tri) = StX
    StX <- (board.counter) assign fifteen = value[0] & value[1] &
        value[2] & value[3]
af.....wire (wire/tri) = StX
    StX <- (board.counter) assign altFifteen = &value
```

The following command lists the drivers of a signal called `top.under_test.sum`.

```
xcelium> drivers top.under_test.sum
top.under_test.sum...output [1:0] (wire/tri) = 2'h0 (-W)
    2'h0 <- (top.under_test) assign {c_out, sum} = a + b + c_in
```

The following command lists the drivers of a signal called `board.count`.

```
xcelium> drivers board.count
board.count.....wire [3:0]
    count[3] (wire/tri) = St1
        St1 <- (board.counter.d) output port 1, bit 0 (.counter.v:10)
    count[2] (wire/tri) = St0
        St0 <- (board.counter.c) output port 1, bit 0 (./counter.v:9)
    count[1] (wire/tri) = St1
        St1 <- (board.counter.b) output port 1, bit 0 (./counter.v:8)
    count[0] (wire/tri) = St0
        St0 <- (board.counter.a) output port 1, bit 0 (./counter.v:7)
```

The following commands list the drivers of a mixed bus `w`.

```
xcelium> drivers w
xmsim: *E,MIXBOFF: Mixed discipline bus 'w' needs an index.
```

```
xcelium> drivers w[0]
xmsim: *E,TNODIA: No drivers exist for analog object: top.w[0].
xcelium> drivers w[1]
w.....wire [0:2]
w[1] (wire/tri) = StX
No drivers
xcelium> drivers w[2]
w.....wire [0:2]
w[2] (wire/tri) = StX
No drivers
```

The following command shows the error message that the simulator displays if you run the `xmsim` simulator in regression mode and then use the `drivers` command to find the drivers of an object that does not have read and connectivity access.

```
xcelium> drivers count
xmsim: *E,OBJACC: Object must have read and connectivity access:
                board.count.
```

The following examples illustrates the output of the `drivers` command when the actual driver is from a VHDL model:

```
xcelium> drivers :u1.a
u1.a.....input (wire/tri) = St1
    St1 <- (:u1) driven by a VHDL model
xcelium> drivers :u1.v.d
u1.v.d.....input (wire/tri) = St1
    St1 <- (:u1) port 'a' in module 'and2' [File: ./verilog.v],
        driven by a VHDL model
xcelium>
```

This report indicates that the signal `:u1.v.d` is ultimately driven by a port (connected to port `a` of the module `and2`) on a module whose body is an imported VHDL model.

Drivers within the scope of automatically inserted connect modules are listed by giving the automatically inserted module name only. The `verilog_source_line_of_the_driver` does not list the source line. For example:

```
xcelium> drivers result
result.....input (wire/tri) = StX
    StX <- (top.analogResultelect_to_logiclogic) module top
```

where `top.analogResultelect_to_logiclogic` is the auto-generated instance name for an auto-inserted connect module

Output for VHDL Signals

The following examples use the VHDL model shown in the "drivers.vhd" section of the "Code Examples" appendix in *Cadence Verilog Simulation User Guide*. A `run` command has been issued after invoking the simulator.

The following command shows the drivers of signal `s`. The string `[verbose report available .....` indicates that type conversion functions or resolution functions are part of the hierarchy of drivers. Use the `-verbose` option to display this additional information.

```
xcelium> drivers s
s.....signal : std_logic = '0'
    [verbose report available.....]
    '0' <- (:GATE:p)
    '0' <- (:) s <= '0' after 1 ns
```

The following command includes the `-novalue` option, which suppresses the display of the current value of each driver.

```
xcelium> drivers s -novalue
s.....signal : std_logic
    [verbose report available.....]
    (:GATE:p)
    (:) s <= '0' after 1 ns
```

The following command includes the `-verbose` option, which causes the inclusion of resolution function and type conversion function information. This report shows that the port `:GATE:q` is one of the contributing drivers, and that there is a type conversion function `bit_to_std` through which the port's value is routed before being assigned to the signal `:s`. The report also shows that there is a concurrent signal assignment statement contributing as one of the sources to the resolution function.

```
xcelium> drivers s -verbose
s.....signal : std_logic = '0'
    '0' <-[resolution function @ieee.std_logic_1164:resolved()]
        <src 1>
            '0' <- (:GATE) [type conversion function
                bit_to_std(<formal>)]
            <formal> connected to port q

                :GATE:q....port : inout BIT = '1'
            '0' <- (:GATE:p)
        <src 2>
            '0' <- (:) s <= '0' after 1 ns
```

The following command shows the drivers `:gate:q`.

```
xcelium> drivers :gate:q
GATE:q.....port : inout BIT = '1'
'0' <- (:GATE:p)
```

The following command includes the `-effective` option, which displays contributions to the effective value of the signal instead of to the driving value.

```
xcelium> drivers :GATE:q -effective

GATE:q.....port : inout BIT = '1'
[verbose report available.....]
'0' <- (:GATE:p)
'0' <- (:) s <= '0' after 1 ns
```

The following command includes the `-verbose` option, which helps you to understand where the effective value of 1 in the previous example comes from.

```
xcelium> drivers :GATE:q -effective -verbose

GATE:q.....port : inout BIT = '1'
'1' <- (:GATE) [type conversion function std_to_bit(<actual>)]
<actual> connected to signal s

:s.....signal : std_logic = '0'
'0' <-[resolution function @ieee.std_logic_1164:resolved()]
  <src 1>
    '0' <- (:GATE) [type conversion function
                  bit_to_std(<formal>)]
    <formal> connected to port q
    :GATE:q.....port : inout BIT = '1'
    '0' <- (:GATE:p)
  <src 2>
    '0' <- (:) s <= '0' after 1 ns
```

The following command shows the output of the `drivers` command when the driver is from a Verilog model.

```
xcelium> drivers -effective il:a
il:a.....port : in std_logic = '1'
'1' <- (and2_top.il) driven by a Verilog model

xcelium> drivers -effective il:il:port1
il:il:port1...port : in std_logic = '1'
'1' <- (and2_top.il) port 'a' in and2(and2_bot) [File:
./and2.vhd], driven by a Verilog model
```

Related Topic

- [drivers Command Report Format](#)

drivers Command Report Format

The `drivers` command supports two report formats based on the following:

- [Verilog Signals](#)
- [VHDL Signals](#)

Verilog Signals

The drivers report for digital Verilog-AMS signals is as follows:

```
value <- ( scope ) verilog_source_line_of_the_driver
```

For example:

```
af.....wire (wire/tri) = St1
    St1 <- (board.counter) assign altFifteen = &value
```

Instead of `verilog_source_line_of_the_driver`, the following is output when the actual driver is from a VHDL model:

```
port 'port_name' in module_name [File:
    path_to_file_containing_module], driven by a VHDL model.
```

This report indicates that the signal is ultimately driven by a port (connected to `port_name` of the specified module) on a module whose body is an imported VHDL model. The `module_name` corresponds to the module name of the shell being used to import the VHDL model.

VHDL Signals

The drivers report for VHDL signals is as follows:

```
description_of_signal = value
    value_contributed_by_driver <- ( scope_name ) source_description
```

The `source_description` for the various kinds of drivers are shown below:

A process	Nothing is generated for the <code>source_description</code> . This implies that a sequential signal assignment statement within a process is the driver. The <code>scope_name</code> is the scope name of the process.

Concurrent signal assignment/concurrent procedure call	The <i>source_description</i> is the VHDL source text of the concurrent signal assignment statement or concurrent procedure call that results in a driving value. This concurrent statement is within the scope <i>scope_name</i> .
No drivers	If the signal has no drivers, the text <code>No drivers</code> appears verbatim.
A Verilog driver	If the driver is from a Verilog model, the report has the following form: <pre>port 'port_name' in entity(arch) [File: path_to_file_containing_entity], driven by a Verilog model.</pre> This report indicates that the signal is ultimately driven by a port (connected to <i>port_name</i> of the specified entity-architecture pair) on an entity whose body is an imported Verilog model.
Driver from a C model	If the driver is from an imported C model, the report has the following form: <pre>port 'port_name' in entity(arch) [File: path_to_file_containing_entity], driven by a C model.</pre>
Driver from a LMC model	If the driver is from an imported LMC model, the report has the following form: <pre>port 'port_name' in entity (arch) [File: path_to_file_containing_entity], driven by a LMC model.</pre>
Driver from an OMI model	If the driver is from an imported OMI model, the report has the following form: <pre>port 'port_name' in entity(arch) [File: path_to_shell_file], driven by a OMI model.</pre>
Resolution/type conversion function in non-verbose Mode	If you do not use the <code>-verbose</code> option, the text <code>[verbose report available]</code> may appear. This indicates that the signal gets its value from a resolution function or a type conversion function. Use <code>-verbose</code> to display more information on the derivation of the signal's value. On the next line of output (indented), a nonverbose driver report is displayed for each signal whose driver contributes to the value of the signal in question.
Resolution function	The following text is generated only when the <code>-verbose</code> option is used: <pre>[resolution function function_name()]</pre> This means that the signal is resolved with the named resolution function. A verbose drivers report is displayed (indented) for all inputs to the resolution function.
Type conversion on formal of port association	The following text is generated only when the <code>-verbose</code> option is used: <pre>[type conversion function function_name(formal)]</pre> This means that the signal's driving value comes from a type conversion function on a <i>formal</i> in a port association. A verbose drivers report is displayed (indented) for the formal port that is the input to the function.

Type conversion on actual of port association	The following text is generated only when the <code>-verbose</code> option is used: <pre>[type conversion function <i>function_name</i>(actual)]</pre> This means that the signal's effective value comes from a type conversion function on an <i>actual</i> in a port association. A verbose drivers report is displayed (indented) for the actual that is the input to the function.
Implicit guard signal	The following text is displayed in response to a query on a signal whose value is computed from a <code>GUARD</code> expression: <pre>[implicit guard signal]</pre>
Signal attribute	The following is displayed in response to a query on an IN port that ultimately is associated with a signal valued attribute: <pre>[attribute of signal <i>full_path_of_the_signal</i>]</pre> The <i>full_path_of_the_signal</i> corresponds to the complete hierarchical path of the signal whose attribute is the driver.
Constant expression on a port association	The following is displayed when the value of the signal in question is from a constant expression in a port map association: <pre>[constant expression associated with port <i>port_name</i>]</pre>
Composite signals	For a composite signal, a separate report is displayed for each group of subelements that can be uniquely named and that have the same set of drivers.

Related Topic

- [drivers](#)

exit

The Tcl `exit` command terminates the simulation and returns control to the operating system.

The [finish](#) command can also exit a simulation.

exit Command Syntax

```
exit  
  
-analogsolver
```

exit Command Options

This section describes the options that you can use with the Tcl `exit` command.

<code>-analogsolver</code>	Exits the analog solver and saves a snapshot of the setup so that you can rework from that checkpoint.

exit Command Examples

The following command ends the simulation session:

```
xcelium> exit
```

The following command lets you exit the analog solver and saves the snapshot of the setup:

```
xcelium> exit -analogsolver
```

finish

The Tcl `finish` command causes the simulator to exit and returns control to the operating system.

finish Command Syntax

```
finish [0 | 1 | 2]
```

finish Command Options

This command takes an optional argument that determines what type of information is displayed.

- 0: Prints nothing (same as executing `finish` without an argument).
- 1: Prints the simulation time. If the analog solver is interactive when the `finish` command is issued, the analog solver's simulation time is printed; otherwise the digital solver's simulation time is printed.
- 2: Prints simulation time as for the argument above and also prints statistics on memory and CPU usage.

finish Command Examples

The following command ends the simulation session and prints the simulation time.

```
xcelium> finish 1
Simulation complete via $finish(1) at time 0 FS + 0
%
```

The following command ends the simulation session, prints the simulation time, and displays memory and CPU usage statistics.

```
xcelium> finish 2
Memory Usage - 7.6M program + 2.1M data = 9.8M total
CPU Usage - 0.9s system + 2.5s user = 3.4s total (28.5% cpu)
Simulation complete via $finish(2) at time 500 NS + 0
%
```

Related Topic

- [Exiting the Simulation](#)

force

The Tcl `force` command sets a specified object to a given value and forces it to retain that value until another force is placed on it or until it is released with:

- A `release` command
- A `force -release` command
- A `deposit -release` command

The new value takes effect immediately, and, in the case of Verilog wires and VHDL signals and ports, the new value propagates throughout the hierarchy before the command returns. Releasing a force causes the value to immediately return to the value that would have been there if the force had not been blocking transactions.

You cannot use the `force` command on an analog object or use the `force` command on digital objects while the analog solver is active.

The object cannot be:

- A Verilog memory
- A Verilog memory element

- A bit-select or part-select of a Verilog register
- A bit-select or part-select of an unexpanded Verilog wire
- A VHDL variable

For Verilog, a force created by the `force` command is identical in behavior to a force created by a Verilog `force` procedural statement. The force can be released by a Verilog `release` statement or replaced by a Verilog `force` statement during subsequent simulation.

The value must be a literal, and the literal is treated as a constant. Even if the literal is generated using value substitution or Tcl's `expr` command, the value is considered to be a constant. The forced value does not change if objects used to generate the literal change value during subsequent simulation.

For VHDL, the value specified with the `force` command must match the type and subtype constraints of the VHDL object. Integers, reals, physical types, enumeration types, and strings (including `std_logic_vector` and `bit_vector`) are supported. Records and non-character array values are not supported, but objects of these types can be assigned to by issuing commands for each subelement individually.

Forces created by the `force` command and those created by a Verilog `force` procedural statements are saved if the simulation is saved.

The `force` command can also be used on wreals with the following limitations:

- The value being forced should be a literal value. The Tcl value substitution (`value x`) and `expr` command (`expr 2+3`) are supported.
- Out of module references (OOMRs) to wreals that require support of Hierarchical IE optimization and IE insertion in VHDL scope are not supported.

You can apply the `force` command on bit-select and part-select of wreal arrays, or on an entire wreal array. In case of part-select, the number of forced values should be equal to the width of part-select. In case of entire wreal arrays, the number of forced values should be equal to the width of the array.

The following net types are supported:

- wreal array, explicitly defined in Verilog-AMS

```
module foo();  
    wreal ra[3:0];  
endmodule
```

- wire vector defined in Verilog-AMS or SystemVerilog file, and coerced to wreal array

```
module top();  
    wire [3:0]ra;  
    foo inst(ra);  
endmodule
```



```
endmodule
```

```
module foo();
    wreal ra[3:0];
endmodule
```

- interconnect vector defined in a SystemVerilog file, and coerced to wreal array

```
module top();
    interconnect [3:0]ra; <<< applying TCL force/release on each bit will be supported
    foo inst(ra);
endmodule
```

```
module foo();
    wreal ra[3:0];
endmodule
```

A wreal net can also be forced to ``wrealXState` and ``wrealZState`.

For an SV-UDN net, the forced value overrides all previous forces on the entire SV-UDN structure. The forced value is frozen, and the resolution function result is overridden always, until another force or release command is applied. The forced value is the final value of net, and no resolution function is executed at that time.

You can apply the `force` command on scalar, whole array, bit-select, or part-select of arrays of SV-UDN. And, you can use order-based or name-based approach to force a value. For the name-based approach, the filed order can be different from that defined in the SV-UDN net. You can also assign `real'z` or `real'x` values to SV-UDN.

A limitation of applying the `force` command on an SV-UDN net is that you cannot force value on individual/specific fields of the SV-UDT. You must specify all the fields of the SV-UDN structure in the `force` command.

Additionally, you can use the `xm_force` system task to assign value on an SV-UDN net. The syntax is: `xm_force ("<name", "value")`. The “value” string should not contain any space; and, for the scalar UDN or bit-selected UDN array, the “value” string should not contain `{}`. For example:

```
$xm_force("dut.ve[0:1]", "{V:1.0,R:10,I:real'z}{1.5,real'x,20}");
```

force Command Syntax

```
force <object-name> [=] <value>
```

force Command Options

None.

force Command Examples

The following examples illustrate use cases for the `force` command:

- [Digital Verilog-AMS](#)
- [VHDL](#)
- [Wreal](#)
- [SV-UDN](#)

Digital Verilog-AMS Examples

The following command forces object `r` to the value ``bx`. The equal sign is optional.

```
xcelium> force r = `bx
```

The following command uses value substitution. Object `x` is forced to the current value of `w`.

```
xcelium> force x = #w
```

The following command uses command substitution and value substitution. Object `y` is forced to the result of the Tcl `expr` command, which evaluates the expression `#r[0] & ~#r[1]` using the current value of `r`.

```
xcelium> force y [expr #r[0] & ~#r[1]]
```

The following command shows the error message that is displayed if you run in regression mode and then use the `force` command on an object that does not have write access.

```
xcelium> force clrb 1
xmsim: *E,RWACRQ: Object does not have read/write access:
                    hardrive.h1.clrb.
```

VHDL Examples

The following command forces object `:t_nickel_out` (`std_logic`) to 1. The equal sign is optional.

```
xcelium> force :t_nickel_out = '1'
```

The following command forces object `:top:DISPENSE_tempsig` (`std_logic`) to 1.

```
xcelium> force :top:DISPENSE_tempsig '1'
```

The following command forces object `:t_dimes` (`std_logic_vector`) to 0.

```
xcelium> force :t_DIMES {"00000000"}
```

The following command forces object `is_ok` (`boolean`) to TRUE.

```
xcelium> force :is_ok true
```

The following command forces object `:count` (`integer`) to 10.

```
xcelium> force :count 10
```

Wreal Examples

The following command applies the `force` command on bit-select of wreal arrays:

```
Time: 100 NS: top.wrvInst.ra = (0,0,0,0)
xcelium> force top.wv[3] 1.2
Time: 100 NS: top.wrvInst.ra = (1.2,0,0,0)
```

The following command applies the `force` command on part-select of wreal arrays:

```
Time: 100 NS: top.wrvInst.ra = (0,0,0,0)
xcelium> force top.wrvInst.ra[2:1] {1.5 1.8}
Time: 200 NS: top.wrvInst.ra = (0,1.5,0,0)
Time: 200 NS: top.wrvInst.ra = (0,1.5,1.8,0)
```

The following command applies the `force` command on an entire wreal array:

```
Time: 100 NS: top.wrvInst.ra = (0,0,0,0)
xcelium> force top.wrvInst.ra {1.8 1.6 1.4 1.2}
Time: 100 FS: top.wrvInst.ra = (1.8,0,0,0)
Time: 100 FS: top.wrvInst.ra = (1.8,1.6,0,0)
Time: 100 FS: top.wrvInst.ra = (1.8,1.6,1.4,0)
Time: 100 FS: top.wrvInst.ra = (1.8,1.6,1.4,1.2)
```

The following command forces the wreal net `top.wrvInst.r` to a value `real'x`.

```
xcelium> force top.wrvInst.r=real'x
```

The following command forces the wreal net `top.wrvInst.r` to a value `real'z`.

```
xcelium> force top.wrvInst.r=real'z
```

The following command forces wreal net `top.wrvInst.ra` to ``wrealXState` and ``wrealZState`.

```
xcelium> force top.wrvInst.ra {real'x 1.6 1.4 real'z}
```

```
Time: 100 NS: top.wrvInst.ra = (`wrealXState,0,0,0)
Time: 100 NS: top.wrvInst.ra = (`wrealXState,1.6,0,0)
Time: 100 NS: top.wrvInst.ra = (`wrealXState,1.6,1.4,0)
Time: 100 NS: top.wrvInst.ra = (`wrealXState,1.6,1.4,`wrealZState)
```

SV-UDN Examples

The following example illustrates the SV-UDN structure and how the `force` command applies a name-based value to a scalar net. Values 1.2, `real'z`, and 500 are forced on real variable V, I, and R, respectively.

```
typedef struct {
  real V;
  real I;
  real R;
} EEstruct;

Time: 0 FS: tb.ve = ('{V:0, I:0, R:1e+06},' {V:0, I:0, R:1e+06},' {V:0, I:0, R:1e+06})
Time: 0 FS: tb.e = '{V:0, I:0, R:1e+06}'

xcelium> force tb.e {V:1.2, I:real'z, R:500};
Time: 40 NS: tb.e = '{V:1.2, I:`wrealZState, R:500}'
```

The following example illustrates how the `force` command applies value on bit-select of an array with order-based value. Values 1.2, `real'z`, and 50 are forced on real variable V, I, and R, respectively in the order in which it is declared in the structure:

```
xcelium> force tb.ve[1] {1.2, real'z, 50}

Time: 40 NS: tb.ve = ('{V:0, I:0, R:1e+06},' {V:1.2, I:`wrealZState, R:50},' {V:0, I:0, R:1e+06})
```

The following example applies the `force` command on part-select of an array:

```
# Part-Selected
xcelium> force tb.ve[2:1] {{V:1.0, R:10, I:real'z} {1.5, real'x, 20}} //named-based and
order-based approach

Time: 60 NS: tb.ve = ('{V:1, I:`wrealZState, R:10},' {V:1.2, I:`wrealZState, R:50},' {V:0, I:0, R:1e+06})

Time: 60 NS: tb.ve = ('{V:1, I:`wrealZState, R:10},' {V:1.5, I:`wrealXState, R:20},' {V:0, I:0, R:1e+06})
```

Related Topics

- [Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)
- [Forcing and Releasing Signal Values](#)
- Basics of Tcl appendix in *Cadence Verilog Simulation User Guide*

get_analog_param

Returns the value for the specified parameter on the Tcl command prompt.

This command is available only when the analog engine is enabled.

get_analog_param Command Syntax

```
get_analog_param  
    -name paramName
```

get_analog_param Command Options

This section describes the options that you can use with the Tcl `get_analog_param` command.

<code>-name</code>	Specifies the name of the parameter whose value is to be displayed.
Example:	
<code>get_analog_param -name iabstol</code>	
Displays the value of the parameter <code>-iabstol</code> .	

Related Topic

- [Appendix B](#)

probe

Controls the values being saved to a database. You can

- Create probes (`-create`)
- Delete probes (`-delete`)
- Disable probes (`-disable`)
- Enable previously disabled probes (`-enable`)
- Create a Tcl script that you can execute to recreate the current databases and probes (`-save`)
- Display information about probes (`-show`)

You cannot use the `probe` command to probe VHDL objects to a [VCD](#) database. You can create a VCD database for VHDL objects by using the `call` command to call predefined CFC routines, which are part of the NC VHDL simulator C interface. See [call](#) for details on the `call` command. See Appendix B, "VCD Format Output," in *NC VHDL Simulator Help* for information on VCD files.

With an [SHM](#) database, you can probe all VHDL signals, ports, and variables that are not declared inside subprograms unless their type falls into one of the following categories:

- Non-standard integer types whose bounds require more than 32 bits to represent
- Physical types
- Access and file types
- Any composite type that contains one of the above types

You can create probes for digital objects only if the objects have read access. (Analog objects have read access by default.) If you specify a digital object as an argument to the `probe` command, and that object does not have read access, the software reports an error. If you specify a scope as an argument to the `probe` command, the software excludes from the probe any digital objects within that scope that do not have read access and reports a warning.

probe Command Syntax

```
probe_command ::=  
    probe probe_params
```

```
probe_params ::=
```

```
[ -create ] ports_to_probe | object create_params ... db_format
| -delete { probe_name | pattern } ...
| -disable { probe_name | pattern } ...
| -enable { probe_name | pattern } ...
| -save [ filename ]
| -show [{ probe_name | pattern } ...] [ -database dbase_name ]
| -emptyok

ports_to_probe ::=
    -all [depth] [{ -flow | domain }] [ -variables ] [ -vspice_cell ]
| -inputs [depth] [{ -flow | domain }]
| -outputs [depth] [{ -flow | domain }]
| -ports [depth] [{ -flow | domain }]

depth ::=
    -depth { n | all | to_cells }

domain ::=
    -domain { analog | digital }

object ::=
    instance_name [{ -flow | domain }]
| port_name [{ -flow | domain }]

create_params ::=
    | -aicms
    | -noaicms
    | -inhconn_signal global_signal
    | -rnmnets
    | -nettype nettype
    | -name probe_name
    | -screen [ -format format_string ] [ -redirect filename ] objects
    | -variables
    | -waveform

db_format ::=
    -shm
| -vcd
| -evcd
| -database dbase_name
```

For detailed syntax information for the `probe` Tcl command, see "probe Command Syntax" in the "probe" section of ["Using the Tcl Command-Line Interface"](#) in the *Xcelium Simulator Tcl Command Reference*.

probe Command Options

This section describes the options that you can use with the Tcl `probe` command.

```
[ -create ]  
[ { object |  
  scope_name } ... ]  
[ options ]
```

Places the values of the specified simulation objects in a database. The simulation objects must have read access.

You can only probe simulation objects that have read access. You can also probe signals inside the SPICE scope. For the list of probe options that are supported in SPICE, see [Probe Options Supported in SPICE Scope](#).

If you are probing an SHM or VCD database, you can add an argument to the `-create` modifier that specifies the following:

- The object or objects you want to trace if you specify `-create` with the `-inhconn_signal` option, the object must be an instance (not a terminal) and must have a full hierarchical name.
- The scope or scopes you want to trace
- A combination of objects and scopes

If you do not specify an argument, the software uses the current debug scope. However, you must include an option that specifies the objects to include in the trace. For more information, see [ports_to_probe](#) in the syntax description for the `probe` command.

If more than one database is open, you must use one of the following options to specify the database into which you want the software to dump the values:

- `-database dbase_name`
- `-shm`
- `-vcd`
- `-evcd`

You can specify the following options for the `-create` modifier when you use the Xcelium simulator with mixed-signal option:

- `-aicms`
- `-noaicms`
- `-all [-variables]`
- `-database dbase_name`
- `-depth {n|all|to_cells}`

The software does not probe currents through inherited connections when processing the `all` argument.

- `-domain {analog|digital}`
- `-emptyok`
- `-evcd`
- `-flow`
- `-inhconn_signal global_signal`
- `-rnmnets`
- `-nettype`
- `-inputs`
- `-name probe_name`
- `-outputs`
- `-ports`
- `-screen [-format format_string] [-redirect filename] objects`
- `-shm`
- `-vcd`

`-aicms`

Probes automatically-inserted connect module (AICM) scopes exclusively. By default, `-aicms` uses `-depth all` for the AICM scope search. You must also specify `-all`, `-domain`, `-variables`, `-ports`, `-outputs`, or `-inputs`.

You cannot use the following options when you specify `-aicms`:

- `-inhconn_signal` `global_signal`
- `-screen` `[-format format_string] [-redirect filename] objects`

Here are some examples:

The following command probes inputs of all AICM scopes relative to the current scope:

```
probe -create -aicms -depth all -inputs
```

Specifying `-depth all` is optional because the default behavior is `-depth all`. So, you could use the following command instead:

```
probe -create -aicms -inputs
```

The following command probes inputs of all AICM scopes relative to the scope `top.i1.i2.i3`:

```
probe -create -aicms -depth all -inputs top.i1.i2.i3
```

Again, you could use the following command instead, because specifying `-depth all` is optional:

```
probe -create -aicms -inputs top.i1.i2.i3
```

`-noaicms`

Uses the `-noaicms` option to exclude the AICM scopes from the probe set specified using the Tcl `probe` command. For example:

```
probe -create -all -noaicms
```

Currently, the `-noaicms` option is applicable for AICMs in Verilog scope only.

`-flow`

Indicates a current probe. You can probe current in Verilog-A and Verilog-AMS modules that are behavioral or that instantiate Spectre subcircuits, Spectre primitives supporting calculated currents, or other Verilog-A or Verilog-AMS modules. You can also probe current in Spectre subcircuit and built-in primitive instances in Verilog-AMS modules. The probed object must be a port or an instance when you use the `-ports` or the `-all` option.

Use `-flow -all` to save port currents and all other values and quantities covered by the `-all` option. You cannot probe currents through inherited connections.

Use `-flow` with `-ports`, `-inputs`, or `-outputs` to probe currents in the specified objects of the scope.

- The software saves waveforms in Verilog modules as `signal _$flow`. Consequently, to plot the signal in SimVision, you need to add the `_$flow` extension to the signal name.
- The software saves waveforms in SPICE or Spectre subcircuits or primitives as just `signal`, without the appended `_$flow`.

You cannot probe currents in

- Simulations using the partition-multirate capability
- AC analysis
- The following Spectre built-in devices: `port`, `delay`, `switch`, `hbt`, `transformer`, `core`, `winding`, `fourier`, `d2a`, `a2d`, `a2ao`, `a2ai`. Use the analog simulation control file to probe the currents in these primitives.

If you create a current probe after the simulation starts, the value of the current becomes available only after the next time period.

`-inhconn_signal`

Specifies the `-inhconn_signal` option in the following format with the `-create` modifier:

```
-inhconn_signal global_signal
```

You must use the `-flow` option when you specify this option. These two options together specify that you want the probe to return the total current drawn from `global_signal` through inherited connections by the specified instance. The `global_signal` must have a full hierarchical name. The software generates a signal named either `global_signal _$flow` or just `global_signal`, depending on whether the instance is of a Verilog module or of a SPICE or Spectre subcircuit or primitive.

`-rnmnets`

Uses the `-rnmnets` to probe only the set of RNM nets in the specified scope. For example:

```
probe -create -shm -rnmnets
```

When you specify the `-rnmnets` option along with `{object, scope_name}`, the tool probes only the RNM nets for the target scope. Similarly, when you use the option with `-aicms, -ports, -inputs, -outputs, or -depth {all, end, to_cell}`, the tool probes only the RNM nets among all the scope/ports.

Using the `-all` option along with the `-rnmnets` option probe all the objects within the given scope; that is, all nets including RNM nets are probed.

You cannot use the following options when you specify `-rnmnets`:

- `-flow`
- `-inhconn_signal global_signal`

`-nettype <nettype>`

Prunes the selection of probes by selecting only a specific nettype. The command configures the simulator to probe a net only when the net belongs to the specified nettype.

When you specify `-nettype` along with the `-depth {all, end, to_cell}` option, the tool controls the levels of hierarchy to be traversed while selecting probes. Optionally, you can specify `{scope_name}` to further prune the probing. If scope is not specified, top level scope is used by default. This supports typical TCL `scope` and/or `-depth` semantics.

To probe multiple nettypes, multiple probe commands must be specified.

`-delete { probe_name
| pattern } ...`

Deletes the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

`-disable {
probe_name | pattern
} ...`

Disables the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

`-enable { probe_name
| pattern } ...`

Resumes the probe(s) specified by the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

`-save [ filename ]`

Creates a Tcl script that you can execute to re-create the current databases and probes. If you do not specify a `filename` argument, the script is printed to the screen.

`-show [{ probe_name |
pattern } ...][-
database dbase_name
]`

Outputs information about the probe(s) specified in the argument. The argument can be a probe name or a list of probe names. You can use the two wildcard characters (`*` and `?`) in the probe name argument.

Add the `-database` sub-option to print information on probe(s) in the specified database.

probe Command Examples

The following command creates a probe on all objects in the current debug scope. All objects have read access. Data is sent to the default **SHM** database. If no default SHM database exists, a default database called `xcelium.shm` in the file `xcelium.shm` is created. The `-create` modifier is not required. The `-all` option (or `-inputs`, `-outputs`, or `-ports`) is required because no `object` or `scope_name` argument is specified.

```
xcelium> probe -create -shm -all
```

The following command creates a probe on all nets of type `mynettype`; and, `mynettype` must be a `nettype`.

```
xcelium> probe -create -shm -nettype mynettype -depth all
```

The following command creates a probe on all nets of type `mynettype` at scope `top.I1.I2` and the level below.

```
xcelium> probe -create -shm -nettype mynettype -depth 2 top.I1.I2
```

The following command creates a probe on all nets of type `mynettype` at the top level.

```
xcelium> probe -create -shm -nettype mynettype
```

The following command creates a probe on all inputs in the current debug scope. Data is sent to the default VCD database. If no default VCD database exists, a default database called `xcelium.vcd` in the file `xcelium.vcd` is created. The `-inputs` option (or `-all`, `-outputs`, or `-ports`) is required because no `object` or `scope_name` argument is specified.

Probing analog objects to a VCD database is not supported, so the following command should not be used for such objects.

```
xcelium> probe -vcd -inputs
```

The following command creates a probe on all ports in the current debug scope. Data is sent to the database waves. This database must already exist. The `-ports` option (or `-all`, `-outputs`, or `-inputs`) is required because no `object` or `scope_name` argument is specified.

```
xcelium> probe -database waves -ports
```

The following command creates a probe on the signal `sum` in the current debug scope and sends data to the default SHM database (creating one called `xcelium.shm` in the file `xcelium.shm`, if necessary).

```
xcelium> probe -shm sum
```

The following command creates a probe on `sum` and `c_out` in the current debug scope and sends data to the default SHM database (creating one called `xcelium.shm` in the file `xcelium.shm`, if necessary).

```
xcelium> probe -shm sum c_out
```

The following command creates a probe on `sum` in scope `u1`, sending data to the default SHM database (creating one called `xcelium.shm` in the file `xcelium.shm`, if necessary).

```
xcelium> probe -shm u1.sum
```

The following command creates a probe on all objects in scope `u1`.

```
xcelium> probe -shm u1
```

The following command creates a probe on all objects in scopes `u1` and `u2`.

```
xcelium> probe -shm u1 u2
```

The following command creates a probe on all ports in scope `u1`.

```
xcelium> probe -shm u1 -ports
```

The following command creates a probe on all ports in scope `u1` and its subsscopes.

```
xcelium> probe -shm u1 -ports -depth 2
```

The following command creates a probe on all ports in scope `u1` and all scopes below `u1`.

```
xcelium> probe -shm u1 -ports -depth all
```

The following command creates a probe on every signal inside spice instance `spice_A1`.

```
xcelium> probe -create -shm top.spice_A1.* -depth all
```

The following command creates a probe on all ports in scope `u1` and all scopes below `u1`, stopping at modules with a ``celldefine` directive.

```
xcelium> probe -shm u1 -ports -depth to_cells
```

The following command creates a probe called `peek`.

```
xcelium> probe -shm sum -name peek
```

The following command monitors value changes on the digital signals `clock` and `count`, and an analog signal, `analogsig`. When either of the digital signals changes value, output for all the signals displays on the screen.

```
xcelium> probe -screen clock count analogsig
Created probe 1
xcelium> run 10 ns
Time: 5 NS: board.clock = 1'h1 : board.count = 4'hx
Ran until 10 NS + 0
```

The following command probes all the port currents of instance `top.A`.

```
probe -create -flow -shm -ports top.A
```

The following command probes a single port current. The command specifies, as it must, the formal port name of the instance being probed.

```
probe -create -shm -flow top.A1.b1
```

The following command probes the current drawn by the `top.A` instance from the `\vdd!` global net through inherited connections. Notice how full hierarchical names are specified for both the instance and the net. The original name of the net is `cds_globals.\vdd!` but preserving that name through the Tcl shell requires a second backslash, the space at the end of the name, and quotation marks.

```
probe -create -shm -flow top.A -inhconn_signal "cds_globals.\vdd! "
```

In the following command, the `-format` option is included to format the output of `probe -screen`.

```
xcelium> probe -screen -format "clock = %d \ncount = %b" clock count
Created probe 1
xcelium> run 10 ns
clock = 1'd1count = 4'bxxxx
Ran until 10 NS + 0
```

The following example illustrates the simulator output when you use `probe -screen` to monitor signal value changes, and then disable the probe at some later time.

```
xcelium> probe -screen clock count
Created probe 1
xcelium> probe -disable 1
xcelium> run 10 ns
Time: 5 NS: board.clock = <disabled> : board.count = <disabled>
Ran until 10 NS + 0
```

The following command displays the state of all probes.

```
xcelium> probe -show
```

The following command displays the state of the probe called `peek`.

```
xcelium> probe -show peek
```

The following command disables the probe called `peek`.

```
xcelium> probe -disable peek
```

The following command enables the probe called `peek`, which was disabled in the previous command.

```
xcelium> probe -enable peek
```

The following command deletes all probes beginning with the characters `pe`.

```
xcelium> probe -delete pe*
```


The following command deletes all probes beginning with the characters `v` and `w`.

```
xcelium> probe -delete {[vw]}
```

The following command shows the error message that is displayed if you run in regression mode and then probe an object that does not have read access.

```
xcelium> probe -shm d
xmsim: *E,RDACRQ: Object does not have read access: harddrive.h1.d.
```

The following command produces a warning message for attempting to probe an analog object to a [VCD](#) database.

```
xcelium> probe -create -vcd top.analogResult -waveform
```

The error message is

```
xmsim: *W,PRALOB: Cannot probe analog object:
    top.analogResult. This object ignored.

xmsim: *E,PRWHAT: no items specified in probe -create command.
```

Related Topics

- [probe](#)
- [Probing SPICE Ports](#)
- [Probing Built-In and User-Defined Nettypes](#)
- [Wildcards Characters in Tcl Commands](#)
- [Access to Digital Simulation Objects](#)

Probing Built-In and User-Defined Nettypes

You can use the `probe -create -screen` and `probe -create -shm` commands to probe objects with built-in and user-defined nettypes and resolution functions. For example, you can use these commands to probe the following:

- Scalar net of type real with built-in and user-defined resolution functions
- Nettypes of type unpacked structure with real, logic, bit, integer and enum data types.
- Unpacked array of nettypes with elements as scalar real with built-in and user-defined resolution functions
- Unpacked array of unpacked structure nettype with built-in and user-defined resolution functions.

Here nettype is defined on the type of the complete structure object and hence, the following different

scenarios are supported:

- Probe on the complete object
- Probe on the member select of complete object

Probe Creation Commands

Probe Commands	Description	
<pre>probe -create -screen < complete_object ¹ ></pre>	Screen probe on object or object member	¹ Object Probes : The probe will be attached on the
<pre>probe -create -screen <c omplete_object.member_select ² ></pre>		
<pre>probe -create -shm < complete_object ¹ ></pre>	Signal probe for shm on object or object member	
<pre>probe -create -shm < complete_object.member_select ² ></pre>		

Probe Manipulation Commands

Probe Commands	Description
<pre>probe -delete <probe_name></pre>	Only removes the respective probe on the specific complete object ¹ or probed member.
<pre>probe -disable <probe_name></pre>	Only disables the respective probe on the specific complete object ¹ or probed member.
<pre>probe -enable <probe_name></pre>	Only enables the respective probe on the specific complete object ¹ or probed member.
<pre>probe -show [<probe_name>]</pre>	Only shows the respective probe on the specific complete object ¹ or probed member. As the probe name is optional, it displays all available probes in case the probe name is not specified.

Related Topics

- [probe](#)
- [User-Defined Nettype and Resolution Function](#)

Probing SPICE Ports

The following table displays the Tcl `probe` command options that can be used to probe signals inside the SPICE hierarchy.

Probe Options Supported in SPICE Scope

<code>-create</code>	<code>-shm</code>	<code>-database dbase_name</code>	<code>-all</code>
<code>-depth</code>	<code>-domain</code>	<code>-emptyok</code>	<code>-exclude</code>
<code>-flow</code>	<code>-name probe_name</code>	<code>-ports</code>	<code>-screen [-format] [-redirect]</code>
<code>-waveform</code>	<code>-delete</code>	<code>-disable</code>	<code>-enable</code>
<code>-save</code>	<code>-show</code>	<code>-vspice_cell</code>	

Tcl probing inside the SPICE hierarchy is not supported on the AIX platform.

Probing SPICE Ports in Verilog/SPICE Boundary

You can use the `-vspice_cell` option with the Tcl `probe` command to probe voltage, current, or both on SPICE ports on Verilog/SPICE or VHDL/SPICE boundary. Consider the following examples:

```
xcelium> probe -database ams_database -vspice_cell
```

The above command performs voltage probe on the SPICE ports on Verilog/SPICE boundary.

You can use the `-flow` and `-all` options with the `-vspice_cell` option, to probe both voltage and current, as follows:

```
xcelium> probe -database ams_database -vspice_cell -flow
```

The above command performs current probe on the SPICE ports on Verilog/SPICE boundary.

```
xcelium> probe -database ams_database -vspice_cell -flow -all
```

The above command performs both voltage and current probes on the SPICE ports on Verilog/SPICE

boundary.

- When you use the `save` or `.probe` commands, then internal SPICE nets as well as SPICE sub-hierarchies are also probed despite using the `-vspice_cell` option. Therefore, you need to either modify or disable all `save` and `.probe` commands in the design to effectively use the `-vspice_cell` option.
- The `-vspice_cell` option takes precedence over other Tcl probes that refer to the same design hierarchy and are ignored.
- Verilog ports are not probed on Verilog/SPICE boundary for SPICE-in-Middle and SPICE-on-Top cases.

Related Topic

- [probe](#)

release

The Tcl `release` command releases any force set on the specified objects. Releasing a force causes the value to immediately return to the value that would have been there if the force had not been blocking transactions.

You cannot use the `release` command on an analog object or while the analog solver is active.

This command releases any force, whether it was created by a `force` command or by a Verilog `force` procedural statement during simulation. The behavior is the same as that of a Verilog `release` statement.

Objects specified as arguments to the `release` command must have write access.

The following objects cannot be forced to a value with the `force` command and, therefore, cannot be specified as the object in a `release` command.

- Memory
- Memory element
- Bit-select or part-select of a register
- Bit-select or part-select of a unexpanded wire
- VHDL variable

You can apply the `release` command on bit-select and part-select of wreal arrays, or on an entire wreal array.

For an SV-UDN net, the target net must be scalar, array, or bit-select and part-select of arrays. A limitation of applying the `release` command on an SV-UDN net is that you cannot release a forced value from individual/specific fields of the SV-UDT.

Additionally, you can use the `xm_release` system task to release the forced value on an SV-UDN net. The syntax is: `xm_release ("<name")`. For example:

```
$xm_release("dut.ve[1:2]");
```

release Command Syntax

```
release [-keepvalue] object_name ...
```

release Command Options

This section describes the options that you can use with the Tcl `release` command.

<code>-keepvalue</code>	Releases the forced object, but retains the forced value.
- You can use wildcard characters in the argument to a `release` command.
- The asterisk (`*`) matches any number of characters.
 - The question mark (`?`) matches any one character.
- However, you cannot use wildcard characters inside escaped names.

release Command Examples

The following command removes a force set on object `x`.

```
xcelium> release x
```

The following command removes a force set on object `:top:DISPENSE_tempsig`.

```
xcelium> release :top:DISPENSE_tempsig
```

The following command releases two objects: `w[0]` and `r`.

```
xcelium> release w[0] r
```

The following command shows what happens if you try to release a force digital variable while the analog solver is active.

```
xcelium> release b6
```

```
*E,SETAIA: Analog engine is active. Cannot release digital object: top.sar.b6
```

The following command applies the release command on bit-select of wreal arrays and then releases the force set on the wreal object:

```
Time: 100 NS: top.wrvInst.ra = (0,0,0,0)
xcelium> force top.wv[3] 1.2
Time: 100 NS: top.wrvInst.ra = (1.2,0,0,0)
xcelium> release top.wrvInst.ra[3]
Time: 200 NS: top.wrvInst.ra = (0,0,0,0)
```

The following command applies the force command on part-select of wreal arrays and then releases the value:

```
Time: 100 NS: top.wrvInst.ra = (0,0,0,0)
xcelium> force top.wrvInst.ra[2:1] {1.5 1.8}
Time: 200 NS: top.wrvInst.ra = (0,1.5,0,0)
Time: 200 NS: top.wrvInst.ra = (0,1.5,1.8,0)
xcelium> release top.wrvInst.ra[1:0]
Time: 300 NS: top.wrvInst.ra = (0,1.5,0,0)
```

The following command applies the force command on an entire wreal array and then releases the value:

```
Time: 100 NS: top.wrvInst.ra = (0,0,0,0)
xcelium> force top.wrvInst.ra {1.8 1.6 1.4 1.2}
Time: 100 FS: top.wrvInst.ra = (1.8,0,0,0)
Time: 100 FS: top.wrvInst.ra = (1.8,1.6,0,0)
Time: 100 FS: top.wrvInst.ra = (1.8,1.6,1.4,0)
Time: 100 FS: top.wrvInst.ra = (1.8,1.6,1.4,1.2)
xcelium> release top.wrvInst.ra
Time: 200 NS: top.wrvInst.r = 0: top.wrvInst.ra = (1.8,1.6,1.4,0)
Time: 200 NS: top.wrvInst.r = 0: top.wrvInst.ra = (1.8,1.6,0,0)
Time: 200 NS: top.wrvInst.r = 0: top.wrvInst.ra = (1.8,0,0,0)
Time: 200 NS: top.wrvInst.r = 0: top.wrvInst.ra = (0,0,0,0)
```

The following example illustrates the SV-UDN structure and how the release command removes a forced value set on a part-select of an array (tb.ve[2:1]):

```
typedef struct {
    real V;
    real I;
    real R;
} EEstruct;
```

```
xcelium> release tb.ve[2:1]
Time: 120 NS: tb.ve = ('{V:1, I:`wrealZState, R:10}', '{V:1.8, I:0, R:200}', '{V:1.8, I:0, R:200}')
Time: 120 NS: tb.ve = ('{V:1.8, I:0, R:200}', '{V:1.8, I:0, R:200}', '{V:1.8, I:0, R:200}')
```

The following command removes a force set on a scalar udn, `tb.e`:

```
xcelium> release tb.e
Time: 140 NS: tb.dut.e = '{V:1.8, I:0, R:200}'
Time: 140 NS: tb.e = '{V:1.8, I:0, R:200}'
```

The following command removes a force set on a bit-select of an array, `tb.ve[0]`:

```
xcelium> release tb.ve[0]
```

Related Topics

- [Access to Digital Simulation Objects](#)
- [Forcing and Releasing Signal Values](#)

reset

The `Tcl reset` command resets the currently loaded model to its original state at time zero. The time-zero snapshot, created by the elaborator, must still be available.

The `reset` command is supported only for pure digital designs and cannot be used for mixed-signal designs.

The Tcl debug environment remains the same as much as possible after a reset.

- Tcl variables remain as they were before the reset.
- [SHM](#) and VCD databases remain open, and probes remain set.

VCD databases created with the `$dumpvars` call in Verilog source code are closed when you reset.

- Breakpoints remain set.
- Watch Windows and the SimVision waveform viewer window remain the same.

Forces and deposits in effect at the time you issue the `reset` command are removed.

reset Command Syntax

`reset`

reset Command Options

None.

reset Command Example

The following command resets the currently loaded model to its original state at time zero. The snapshot created at time zero must still be available.

```
xcelium> reset
```

This command does not work on mixed-signal designs. The following example shows what happens if you try to reset a mixed-signal design.

```
xcelium> reset
```

```
*E,RESTAG: Reset not supported for mixed-signal designs.
```

restart

The Tcl `restart` command replaces the currently simulating snapshot with another snapshot of a same design. The simulator then uses the analog control options associated with the new snapshot to continue the simulation.

The specified snapshot must be a snapshot created by the `save` command.

The software interprets the snapshot name the same way as the snapshot name on the `xmsim` command line, with the addition that, if you want, you can give only the view name preceded by a colon to load a snapshot that is a view of the currently loaded cell. For example:

<code>restart top</code>	Restarts <code>[lib .] top [: view]</code> If the view name is omitted, there must be only one snapshot of the given cell, otherwise the snapshot name is ambiguous. In this case, an error message is issued, and a list of available snapshots is printed.
<code>restart top:ckpt</code>	Restarts <code>[lib .] top:ckpt</code>

restart :ckpt	Restarts [<i>lib .</i>][<i>cell</i>]:ckpt

When you restart with a saved snapshot in the same simulation session:

- SHM databases remain open and all probes remain set.
- Breakpoints set in the current session at the time that you execute the restart remain set. Breakpoints are not saved in snapshots.

After a restart, periodic breakpoints trigger at the same times they would trigger without a save and restart. This is true even when the save and restart takes place at a time in between the periodic breakpoints.

- Probes set in the current session at the time you execute the restart remain set. Probes are not saved in snapshots.
- Forces and deposits in effect at the time you issue a `save` command are still in effect when you restart.

If you exit the simulation and then start the simulator with a saved snapshot, databases are closed. Any probes and breakpoints are deleted. If you want to restore the full Tcl debug environment when you restart, make sure that you save the environment with the `save -environment` command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then use the Tcl `source` command after restarting or the `-input` option when you start the simulator to execute the script. For example,

```
xmsim top
    (open a database, set probes, set breakpoints, deposits, forces, etc.)
xcelium> run 100 ns
xcelium> save worklib.top:ckpt1
xcelium> save -environment ckpt1.tcl
xcelium> exit
xmsim -tcl worklib.top:ckpt1
xcelium> source ckpt1.tcl
```

restart Command Syntax

```
restart <snapshot_name>
[-path <snapshot_dir>]
[-show]
```

restart Command Options

This section describes the options that you can use with the Tcl `restart` command.

<code>-path snapshot_dir</code>	Restarts the simulation with the snapshot saved at the specified location. This option lets you restart with a snapshot that was saved to a specific location with a <code>save -path</code> command. For example: <pre>xcelium> save -path /path/to/new/snapshot/dir worklib.top:ckpt1 xcelium> [some Tcl commands] xcelium> restart -path /path/to/new/snapshot/dir worklib.top:ckpt1</pre> This <code>restart</code> command first reads in the library at the location specified by the <code>-path</code> option, and then loads the specified snapshot.
<code>-show</code>	Lists the names of all snapshots that can currently be used as the argument to the <code>restart</code> command.

restart Command Limitations

The following restrictions apply when using the `restart` command.

- You cannot restart a snapshot that is topologically different from the currently loaded snapshot. The simulator determines if snapshots are topologically different by comparing the analog control options saved in the restarted snapshot with the analog control options being used in the current session. Topological differences arise when the `useprobes` parameter of the immediate `set options` statement is set to `yes` and there is a change in the value of any of the following:
 - `options save`
 - `options currents`
 - `options subcktprobelvl` Topological differences can also arise if you attempt to restart a design that is different from the design that is running in the current session. When you do need to load a topologically different snapshot, exit `xmsim` and then start it with the new snapshot.
- To ensure accuracy, the analog simulation control options used to simulate a restarted snapshot must be the same as the options in use when the snapshot is saved.
- You cannot use the `restart` command after the current simulation session has finished.
- If your design is a mixed-signal design, you cannot simulate snapshots saved before the most recent elaboration of the design.

restart Command Examples

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. This snapshot can then be loaded using either of the next two `restart` commands.

```
xcelium> save top:ckpt1
xcelium> restart top:ckpt1
xcelium> restart :ckpt1
```

In the following example, a `save` command is issued to save the simulation state as a view of the currently loaded cell, `top`. A second `save` command is issued to save the Tcl debug environment. If you exit the simulator, you can restart with the saved snapshot and then restore the debug settings by sourcing the script created with the `save -environment` command.

```
xcelium> save top:ckpt1
xcelium> save -environment top_ckpt1.env
xcelium> exit
xcelium -tcl :ckpt1
xcelium> source top_ckpt1.env
```

The following command reloads the snapshot of the given cell, `top`. Because the view name is not specified, the snapshot name is ambiguous if there is more than one view, and an error message is issued.

```
xcelium> restart top
```

The following command lists all of the snapshots you can currently load with the `restart` command.

```
xcelium> restart -show
otherlib.board:module
worklib.board:ckpt1
worklib.board:ckpt2
```

Related Topic

- [How to Use save, restart, and analog](#)

run

The Tcl `run` command starts simulation or resumes a previously halted simulation.

run Command Syntax

```
run
    -analogsolver
    -clean
    -delta [cycle_spec]
    -next
    -phase
    -process
    -return
    -step
    -sync
    [-timepoint] [time_spec] [-absolute | -relative]
```

run Command Options

This section describes the options that you can use with the Tcl `run` command.

<code>-analogsolver</code>	Reboots the analog solver to the exact setup you left at during the last run.
<code>-clean</code>	Runs the simulation to the next point at which it is possible to create a checkpoint snapshot with the <code>save -simulation</code> command. See save .
<code>-delta</code> <code>[cycle_spec]</code>	Runs the simulation for the specified number of delta cycles. If no <code>cycle_spec</code> argument is specified, run the simulation to the beginning of the next delta cycle. A <code>run -delta</code> command is the same as <code>run -delta 1</code> .
<code>-next</code>	Runs one line of source code, stepping over any subprogram calls. If the current execution point is a VHDL non-zero <code>wait</code> statement, <code>run -next</code> might behave the same as <code>run -step</code> . For example, if the current execution point is a <code>wait</code> statement, which suspends the current process, another process might be scheduled to run at the current simulation time. In this situation, <code>run -next</code> runs the next behavioral statement, and the simulation stops in the scheduled process. If you want to run to the next executable line in the source code after the <code>wait</code> , set a line breakpoint on the line and enter a <code>run</code> command.
<code>-phase</code>	Runs to the beginning of the next phase of the simulation cycle. A simulation cycle consists of two phases: signal evaluation and process execution.
<code>-process</code>	Runs until the beginning of the next scheduled process or to the beginning of the next delta cycle, whichever comes first. In VHDL, a process is a process statement. In Verilog it is an <code>always</code> block, an <code>initial</code> block, or some other behavior that can be scheduled to run. For the purposes of <code>run -process</code> , the <code>analog</code> block is not considered a process.
<code>-return</code>	Runs until the current subprogram (task, function, procedure) returns.

<code>-step</code>	Runs one behavioral statement, stepping into subprogram calls. The <code>-step</code> option does not step into function calls made by an <code>analog</code> statement. In this situation, the behavior of the <code>-step</code> option is identical to the behavior of the <code>-next</code> option.
<code>-sync</code>	Runs until the analog solver next hands control to the digital solver.
<code>[-timepoint]</code> <code>[time_spec] [-</code> <code>absolute -</code> <code>relative]</code>	Runs until the specified time is reached. The time specification can be absolute or relative. Relative is the default. In addition to time units such as fs, ps, ns, us, and so on, you can use <code>deltas</code> as the unit. For example: <pre>xcelium> run 10 deltas</pre> This is the same as <code>run -delta 10</code>. If you include a time specification and a breakpoint or interrupt stops simulation before the specified time is reached, the time specification is thrown away. For example, in the following sequence of commands, the last <code>run</code> command does not stop the simulation at 500 ns. <pre>xcelium> stop -object x Created stop 1 xcelium> run 500 ns Stop 1 {x = 0} at 10 ns xcelium> run</pre> Using <code>run -timepoint</code> without a <code>time_spec</code> argument runs the simulation until the next scheduled event.

run Command Examples

The following command runs the simulation until an interrupt occurs or until simulation completes.

```
xcelium> run
```

The following command advances the simulation to 500 ns absolute time. The `-timepoint` option is not required.

```
xcelium> run -timepoint 500 ns -absolute
```

The following command advances the simulation 500 ns relative time. With a time specification, `relative` is the default.

```
xcelium> run 500 ns
```

The following command runs one behavioral statement, stepping into any subprogram calls.

```
xcelium> run -step
```

The following command runs one behavioral statement, stepping over any subprogram calls.

```
xcelium> run -next
```

The following command runs until the current subprogram returns. The subprogram can be a task, function, or procedure.

```
xcelium> run -return
```

The following two commands are equivalent. They both run the simulation for 5 delta cycles.

```
xcelium> run -delta 5  
xcelium> run 5 deltas
```

The following command runs the simulation until the digital solver next becomes active.

```
xcelium> run -sync  
Ran until 2 NS + 0
```

The following command reboots the analog solver to the exact setup you left at during the last run.

```
xcelium> run -analogsolver
```

Related Topic

- [Starting or Resuming a Simulation](#)

save

The Tcl `save` command creates a snapshot of the current simulation state. You can then use the `restart` command to load the saved snapshot and resume simulation.

You may only issue the `save` command when the simulator is at certain points in its execution cycle.

- The simulator cannot be in the middle of executing procedural statements. To run the simulation to the next point at which the `save` command will work, use the `run -clean` command.
- For mixed-signal designs, the simulator must have accepted at least one transient time step.

If your files are very large, you might encounter a limit on the size of a file. If a library database exceeds this limit, you cannot add objects to the database. If you save many snapshot checkpoints to unique views in a single library, this file size limit could be exceeded. If you reach this limit, you can:

- Use `save -overwrite` to overwrite an existing snapshot. For example,

```
xcelium> save -simulation -overwrite snap1
```

- Save snapshots to a separate library. For example,

```
mkdir xcelium.d/snaplib
```

```
xmsim -f xmsim.args
```

```
xcelium> run 1000 ns
```

```
xcelium> save -simulation snaplib.snap1
```

```
xcelium> run 1000 ns
```

```
xcelium> save -simulation snaplib.snap2
```

- Remove snapshots using the `xmrm` utility. For example,

```
xmrm -snapshot worklib.snap1
```

save Command Syntax

```
save -simulation <snapshot_name>  
    [-overwrite]  
    -environment [filename]  
    -commands [filename]
```

save Command Options

This section describes the options that you can use with the Tcl `save` command.

<code>-simulation snapshot_name</code>	Creates a snapshot of the current simulation state. The snapshot also contains the analog control options in effect at the time of the save. This option is the default. You must specify a snapshot name with the <code>save</code> command. The snapshot name can be specified using the <code>[lib.]cell[:view]</code> notation, or, if you want the snapshot to be a new view of the currently loaded cell, you can specify just the view name preceded by a colon. For example, if you are simulating <code>worklib.top:rtl</code>, • <code>save ckpt1</code> Saves <code>worklib.ckpt1:rtl</code> • <code>save top:ckpt1</code> Saves <code>worklib.top:ckpt1</code> • <code>save otherlib.top</code> Saves <code>otherlib.top:rtl</code> • <code>save :ckpt1</code> Saves <code>worklib.top:ckpt1</code> The snapshot name must be a simple name containing only letters, numbers, and underscores. By default, the saved snapshot is saved either in the library you specify or in the work library (if you do not specify a library).
<code>-overwrite</code>	Overwrites an existing snapshot.
<code>-environment [filename]</code>	Creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. The <i>filename</i> argument is optional. If no filename is specified, the script is written to standard output. The state of the Tcl debug environment is not part of the simulation that is saved in a snapshot. To save the debug environment, you must issue a separate <code>save -environment</code> command. This command creates a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. You can then restore the environment by executing this script with the Tcl <code>source</code> command, or you can use the <code>-input</code> option when you start the simulator. For example: <pre>xcelium> save :ckpt1 xcclium> save -environment ckpt1.tcl xcclium> restart :ckpt1 xcclium> source ckpt1.tcl</pre> or <pre>xcelium> -tcl cell:ckpt1 -input ckpt1.tcl)</pre> These scripts are meant to be sourced into an empty environment (that is, an environment with no breakpoints, no probes, no databases). If you start the simulator, set some breakpoints and probes, and then source a script that contains commands to set breakpoints and probes, the simulator will probably generate errors telling you that some commands in the script could not be executed. These errors are due to name conflicts. For example, you may have set a breakpoint that received the default name "1", and the command in the script is trying to create a breakpoint with the same name. You can, of course, give your breakpoints unique names to avoid this problem. You can also edit the scripts to make them work the way you would like them to work.
<code>-commands [filename]</code>	<code>save -commands</code> is the same as <code>save -environment</code>.

save Command Examples

The section includes the examples of how the `save` command in the following scenarios:

- [How to Save a Snapshot of the Current Simulation State](#)
- [How to Use the `save -environment` Command](#)
- [How to Use `save`, `restart`, and `analog`](#)

How to Save a Snapshot of the Current Simulation State

The following command saves the simulation state in `lib . cell :ckpt1`, where `lib` is the name of the current work library, and `cell` is the cell name of the currently loaded snapshot.

```
xcelium> save -simulation :ckpt1
```

The following command saves the simulation state in `lib .top:ckpt1`.

```
xcelium> save top:ckpt1
```

The following command saves the simulation state in `lib .ckpt1: view_name`, where `view_name` is the view name that is currently being simulated.

```
xcelium> save ckpt1
```

How to Use the `save -environment` Command

The following example illustrates how to use the `save -environment` command. The right column annotates the behavior of each command.

Commands	Notes
<pre>xcelium -tcl hardrive xmsim: v2.1.(p1): (c) Copyright 1995 - 2003 Cadence Design Systems Loading snapshot worklib.hardrive:module Done</pre>	Start the simulator.
<pre>xcelium> stop -create -line 32 Created stop 1 xcelium> stop -create -object hardrive.clkCreated stop 2 xcelium> probe -create -shm hardrive.data Created default SHM database xcelium.shm Created probe 1</pre>	Set a line breakpoint, an object breakpoint, and create a probe. The probe command creates a default SHM database.

<pre>xcelium> run 0 FS + 0 (stop 2: hardrive.clk = 0) ./hardrive.v:13 clk = 0;</pre>	
<pre>xcelium> run 50 NS + 0 (stop 2: hardrive.clk = 1) ./hardrive.v:16 always #50 clk = ~clk;</pre>	
<pre>xcelium> save -environment env1.env</pre>	Save the debug settings in a file called <code>env1.env</code>.
<pre>xcelium> more env1.env set assert_report_level {note} set assert_stop_level {error} set autoscope {yes} set display_unit {auto} set tcl_prompt1 {puts -nonewline "xmsim> "} set tcl_prompt2 {puts -nonewline "> "} set time_unit {module}set vlog_format {%h} set assert_1164_warnings {yes} stop -create -name 1 -line 32 hardrive stop -create -name 2 -object hardrive.clk database -open -shm -into xcelium.shm xcelium.shm -defaultprobe -create -name 1 -database xcelium.shm hardrive.datascope -set hardrive</pre>	Look at the file contents.
<pre>xcelium> exit</pre>	Exit from the simulator.
<pre>foghorn% xmsim -tcl hardrivexcelium: v1.2.(b9): (c) Copyright 1995 - 2000 Cadence Design Systems Loading snapshot worklib.hardrive:module Done</pre>	Restart the simulator.
<pre>xcelium> stop -showNo stops set</pre>	The new session has no breakpoints.
<pre>xcelium> source env1.env</pre>	Source the <code>env1.env</code> file.
<pre>xcelium> stop -show1 Enabled Line: ./hardrive.v:32 (scope: hardrive)2 Enabled Object hardrive.clkxcelium> probe -show1 Enabled hardrive.data (database: xcelium.shm) -shmxcelum> database -showxcelum.shm Enabled (file: xcelium.shm) (SHM) (default)</pre>	Show the status of breakpoints, probes, and databases.

How to Use save, restart, and analog

You can use the `save`, `restart`, and `analog` commands during transient analysis of a mixed-signal simulation in interactive mode (under `-tcl` or `-gui`). In non-interactive mode or if there is other analysis after the transient analysis specified analog control file, these commands are ignored when the simulation has reached the analog stop time of the transient analysis.

The `analog -stop` option is ignored with a warning message when `.probe` is used in the analog control file.

The following example illustrates how to use the `save`, `restart` and `analog` commands in interactive mode. The right column annotates the behavior of each command.

Commands	Notes
<code>xcelium> database -open waves -into waves.shm -default</code> Created default SHM database waves	Open a default SHM database called <code>waves</code> .
<code>xcelium> probe -create -all</code> Created probe 1	Probe all signals in the current scope.
<code>xcelium> stop -create -time -absolute 200ns</code> Created stop 1	Create a breakpoint at absolute time 200ns.
<code>xcelium> run</code> 200 NS + 0 (stop 1)	Run until 200ns.
<code>xcelium> analog -show</code> vabstol = 1.000000e-06 (alterable) iabstol = 1.000000e-12 (alterable) reltol = 1.000000e-03 (alterable) stop = 14 us (alterable)	Examine the tolerance values and stop time.
<code>xcelium> save :ckpt1</code> Saved snapshot amslib.top:ckpt1	Save the first snapshot
<code>xcelium> analog -stop 20 us</code>	Lengthen the simulation.
<code>xcelium> analog -reltol 5e-03</code>	Change the <code>reltol</code> value.
<code>xcelium> analog -show</code> vabstol = 1.000000e-06 (alterable) iabstol = 1.000000e-12 (alterable) reltol = 5.000000e-03 (alterable) stop = 20 us (alterable)	Examine the new values.

<pre>xcelium> stop -create -time -absolute 400ns Created stop 2</pre>	Create another breakpoint.
<pre>xcelium> run 400 NS + 0 (stop 2)</pre>	Run until 400ns.
<pre>xcelium> save :ckpt2 Saved snapshot amslib.top:ckpt2</pre>	Save the second snapshot.
<pre>xcelium> restart :ckpt1 Loaded snapshot amslib.top:ckpt1</pre>	Reload the first snapshot.
<pre>xcelium> analog -show vabstol = 1.000000e-06 (alterable) iabstol = 1.000000e-12 (alterable) reltol = 1.000000e-03 (alterable) stop = 14 us (alterable)</pre>	Check that tolerance values and stop time are the same values you had originally...
<pre>xcelium> time 200 NS</pre>	...and that you are back at 200ns in the simulation.
<pre>xcelium> restart :ckpt2 Loaded snapshot amslib.top:ckpt2</pre>	Now reload the second snapshot.
<pre>xcelium> analog -show vabstol = 1.000000e-06 (alterable) iabstol = 1.000000e-12 (alterable) reltol = 5.000000e-03 (alterable) stop = 20 us (alterable)</pre>	Check for the changed <code>reltol</code> value and the lengthened simulation.
<pre>xcelium> time 400 NS</pre>	Run to 400ns, again!
<pre>xcelium> run 200 NS Ran until 600 NS + 0</pre>	Run another 200ns.
<pre>xcelium> restart :ckpt1 Loaded snapshot amslib.top:ckpt1</pre>	Switch back to the first snapshot.
<pre>xcelium> time 200 NS</pre>	Check where you are.

<pre>xcelium> run The analog simulator has reached stop time, please use analog -stop <new stop time> to extend the analog stop time. Simulation complete via transient analysis stoptime at time 14 US Memory Usage - 19.4M program + 13.5M data = 32.9M total CPU Usage - 0.6s system + 3.6s user = 4.2s total (1.1% cpu)</pre>	Run until analog stop time.
<pre>xcelium> analog -stop 100us</pre>	At analog stop time, you can still lengthen the simulation.
<pre>xcelium> run The analog simulator has reached stop time, please use analog -stop <new stop time>' to extend the analog stop time. Simulation complete via transient analysis stoptime at time 100 US Memory Usage - 19.4M program + 13.5M data = 32.9M total CPU Usage - 0.6s system + 23.6s user = 24.2s total (1.1% cpu)</pre>	Run until analog stop time.
<pre>xcelium> restart :ckpt1 Loaded snapshot amslib.top:ckpt1</pre>	At analog stop time, you can still restart a snapshot.
<pre>xcelium> exit Number of accepted tran steps = 3161. Initial condition solution time = 200 ms. **** AMSD: Mixed-Signal Activity Statistics **** Number of A-to-D events: 24 Number of A-to-D events in IEs: 0 Number of D-to-A events: 16 Number of D-to-A events in IEs: 0 Number of VHDL-AMS Breaks: 0 Intrinsic tran analysis time = 3.5 s. Total time required for tran analysis tran1 was 23.7 s.</pre>	Exit and complete the simulation.

Related Topics

- [Controlling the Simulation in SimVision](#)
- [Mixed-Signal Design Simulation](#)

scope

Lets you

- Set the current debug scope (`-set`)
- List the automatically-inserted connect module instances within a scope or branch of the design hierarchy (`-aicms`)
- Describe items declared within a scope (`-describe`)
- Display the drivers of digital objects declared within a scope (`-drivers`)
- List the resolved disciplines of all nets within a scope or branch of the design hierarchy (`-disciplines`)
- Print the source code, or part of the source code, for a scope (`-list`)
- Display scope information (`-show`)

In this release, you cannot set scope into an auto-inserted connect module instance in a mixed-signal design. Nor can you describe such a scope, or list its drivers or source lines.

See the "Traversing the Model Hierarchy" section of the "Debugging Your Design" chapter in *Cadence Verilog Simulation User Guide* for more information.

scope Command Syntax

```
scope [-set] [scope_name]
      -up
      -aicms [scope_spec]
            -recurse
            -all
      -describe [scope_name]
            -names
            -sort {name|kind|declaration}
      -drivers [scope_name]
      -disciplines [scope_spec]
            -recurse
            -all
            -sort {name|kind|declaration}
      -list [line|start_line end_line] [scope_name]
      -show
```

scope Command Options

This section describes the options that you can use with the Tcl `scope` command.

`-set [scope_name]` Sets the current debug scope to the specified scope. If no scope or other option is given, the name of the current scope is printed.

The `-set` modifier is optional.

`-up` Sets the debug scope to one level up the hierarchy from the current scope.

`-aicms [scope_spec]` Lists automatically-inserted connect modules (AICMs) inserted within the specified scope, or within the current debug scope if no scope is specified.

`-recurse -all`

The `-recurse` option descends recursively through the design hierarchy, starting with the specified scope (or the current debug scope if no scope is specified), listing all the AICM instances.

The `-all` option lists the AICM instances in all top-level scopes. If used with `-recurse`, the `-all` option recursively lists all AICM instances in the entire design.

<pre>- describe [scope_name]] -names -sort {name kind declaration}</pre>	Describes all objects declared within the specified scope. If no scope is specified, objects in the current debug scope are described. For objects without read access, the output of <code>scope -describe</code> does not include the object's value. For objects that have read access but no write access, the string <code>(-W)</code> is included in the output. For objects with neither read nor write access, the string <code>(-RW)</code> is included in the output. The <code>-names</code> option displays only the names of each declared item in the scope. The <code>-sort</code> option specifies the sort order. There are three possible arguments to the <code>-sort</code> option: • name: sort alphabetically by name • kind: sort by declaration type (reg, wire, instance, branch, etc.) • declaration: sort by the order in which objects are declared in the source code
<pre>- drivers [scope_name]</pre>	Shows the drivers of each digital object declared within the specified scope. If no scope is specified, the drivers of digital objects in the current debug scope are displayed. The output of <code>scope -drivers</code> includes only the digital objects that have read access. However, even if an object has read access, its drivers may have been collapsed, combined, or optimized away, and the output of the command might indicate that the object has no drivers.
<pre>- disciplines [scope_name]] -recurse -all -sort { name kind declaration}</pre>	Lists all resolved net disciplines within the given scope, or within the current debug scope if no scope is given. The <code>-recurse</code> option descends recursively through the design hierarchy, starting with the specified scope (or the current debug scope if no scope is specified), listing all resolved net disciplines. The <code>-all</code> option lists all resolved net disciplines in all top-level scopes. If used with <code>-recurse</code>, recursively lists all resolved net disciplines in the entire design. The <code>-sort</code> option sorts the nets alphabetically by net name, by discipline (<code>electrical</code>, <code>logic</code>, and so on) or by the order they are declared in the source code. The default is to sort by discipline.
<pre>-list [line start_line end_line] [scope_name]</pre>	Prints lines of source code for the specified scope, or for the current debug scope if no scope is specified. You can follow the <code>-list</code> modifier with • No range of lines to print all lines for the scope. • One line number to display that line of source text. • Two line numbers to display the text between those two line numbers. You can use a dash (-) for either <code>start_line</code> or <code>end_line</code>.
<pre>-show</pre>	Shows scope information, including the current debug scope, instances within the debug scope, and top-level modules in the currently loaded model.

scope Command Example

The following example prints the name of the current scope. The `-set` modifier is not required.

```
xcelium> scope -set
```

The following example sets the debug scope to scope `u1`. The `-set` modifier is not required.

```
xcelium> scope -set u1
```

The following example moves the debug scope up one level in the hierarchy.

```
xcelium> scope -up
```

For the next example, you have a design that contains a top level module (`top`) in which three `connect_module` instances are instantiated with a merged connect mode attribute.

The command

```
xcelium> scope -aicms -all -recurse
```

lists all automatically-inserted connect module (AICM) instances in the design as follows.

```
top.connect5a__elect_to_logic__logic (merged) is:
  instance of connect_module:      elect_to_logic,
  inserted across signal:          top.connect5a,
  and ports of discipline:         logic.
top.connect0a__elect_to_logic__logic (merged) is:
  instance of connect_module:      elect_to_logic,
  inserted across signal:          top.connect0a,
  and ports of discipline:         logic.
top.connect2a__elect_to_logic__logic (merged) is:
  instance of connect_module:      elect_to_logic,
  inserted across signal:          top.connect2a,
  and ports of discipline:         logic.
```

The following example shows the output of a similar design, in which the value of the connect mode attribute is `split`.

```
xcelium> scope -aicms -all -recurse

top.connect5a__dig5__in (split) is instance of connect_module elect_to_logic:
  connected where signal:          top.connect5a,
  joins port:                      in,
  of instance:                     dig5.
top.connect5a__dig6__in (split) is instance of connect_module elect_to_logic:
  connected where signal:          top.connect5a,
  joins port:                      in,
  of instance:                     dig6.
top.connect0a__dig0__in (split) is instance of connect_module elect_to_logic:
```

```
connected where signal:      top.connect0a,
joins port:                  in,
of instance:                 dig0.
top.connect0a__dig1__in (split) is instance of connect_module elect_to_logic:
connected where signal:      top.connect0a,
joins port:                  in,
of instance:                 dig1.
```

The following example illustrates how to display a list of resolved disciplines.

```
xcelium> scope -discipline -recurse
net disciplines for: top.I3 (sareg)

result.....input (logic)
clkSig.....input (unknown discipline)
trigger....input (unknown discipline)

net disciplines for: top.I4 (daconv)

compSig....output (electrical)
b0.....input (logic)
b1.....input (logic)
b2.....input (logic)
b3.....input (logic)
b4.....input (logic)
b5.....input (logic)
b6.....input (logic)
b7.....input (logic)

net disciplines for: top.I0 (signalSrc)

gnd.....analog net (electrical)
sig.....output (electrical)

net disciplines for: top.I2 (comparator)

inn.....input (electrical)
inp.....input (electrical)
net55.....wire (electrical)
net79.....wire (electrical)
net84.....wire (electrical)
net92.....wire (electrical)
net94.....wire (electrical)
out.....output (electrical)
vref1.....wire (electrical)

net disciplines for: top.I1 (samplehold)

gnd.....analog net (electrical)
```

```
holdSig....output (electrical)
inSig.....input (electrical)
trigger....input (unknown discipline)

net disciplines for: top.compOut__elect_to_logic__logic (elect_to_logic)

aVal.....input (electrical)
dVal.....output (logic)
```

The following command displays the disciplines of nets and buses, one of which is a mixed bus.

```
xcelium> scope -discipline
net disciplines for: top (top)
e.....wire (electrical)
d.....wire (unknown discipline)
w.....wire (mixed bus)
```

The following command displays a list and a description of all objects declared in the current debug scope (a Verilog-AMS module).

```
xcelium> scope -describe
clr.....register = 1'hx
clk.....register = 1'hx
data.....register [3:0] = 4'hx
q.....wire [3:0] (wire/tri) = 4'hx
end_first_pass...named event
h1.....instance of module hardreg
inSig.....analog net (electrical) = 3.45
vplus5_ground....branch(vplus5) = 2.22
sig1.....inout (electrical) = 0.12
R1.....instance of 'resistor' Spice primitive
vout_vspply_n....branch(vout,vspply_n) = 0
```

The following command displays a list and a description of all objects declared in the current debug scope (a VHDL architecture).

```
xcelium> scope -describe
top.....component instantiation
load_nickels....process statement
load_dimes.....process statement
load_cans.....process statement
load_action.....process statement
gen_clk.....process statement
gen_reset.....process statement
gen_nickels.....process statement
gen_dimes.....process statement
gen_quarters....process statement
$PROCESS_000....process statement
```

```
$PROCESS_001.....process statement
stoppit.....signal : BOOLEAN = TRUE
t_NICKEL_OUT.....signal : std_logic = '0'
t_EMPTY.....signal : std_logic = '1'
t_EXACT_CHANGE...signal : std_logic = '0'
t_TWO_DIME_OUT...signal : std_logic = 'Z'
...
...

t_NICKELS.....signal : std_logic_vector(7 downto 0) = "11111111"
t_RESET.....signal : std_logic = '0'
```

The following command lists the names of all objects declared in the current debug scope. No description is included.

```
xcelium> scope -describe -names
clr clk data q end_first_pass h1
```

The following example displays a list and a description of all objects declared in the current debug scope. Objects are listed in alphabetical order.

```
xcelium> scope -describe -sort name
```

The following command displays a list and a description of all objects declared in the current debug scope. Objects are sorted by type of declaration.

```
xcelium> scope -describe -sort kind
```

The following example displays a list and a description of all objects declared in scope `h1`. Objects are listed in the order in which they were declared in the source code.

```
xcelium> scope -describe -sort declaration h1
clk.....input (wire/tri) = StX
clrb.....input (wire/tri) = StX
d.....input [3:0] (wire/tri) = 4'hx
compSig.....output (electrical) = 0
q.....output [3:0] (wire/tri) = 4'hx
f1.....instance of module flop
f2.....instance of module flop
f3.....instance of module flop
f4.....instance of module flop
compSig_ground..branch(compSig) = 0
```

The following command shows the drivers for all objects declared in scope `h1`.

```
xcelium> scope -drivers h1

clk.....input (wire/tri) = St1
    St1 <- (hardrive.h1) input port 2, bit 0 (./hardrive.v:8)
```

```

clrb.....input (wire/tri) = St1
    St1 <- (hardrive.h1) input port 3, bit 0 (./hardrive.v:8)
d.....input [3:0] (wire/tri) = 4'h2
    [3] = St0
        St0 <- (hardrive.h1) input port 1, bit 3 (./hardrive.v:8)
    [2] = St0
        St0 <- (hardrive.h1) input port 1, bit 2 (./hardrive.v:8)
    [1] = St1
        St1 <- (hardrive.h1) input port 1, bit 1 (./hardrive.v:8)
    [0] = St0
        St0 <- (hardrive.h1) input port 1, bit 0 (./hardrive.v:8)
q.....output [3:0] (wire/tri) = 4'h1
    [3] = St0
        St0 <- (hardrive.h1.f4) nd7 (q, e, qb)
    [2] = St0
        St0 <- (hardrive.h1.f3) nd7 (q, e, qb)
    [1] = St0
        St0 <- (hardrive.h1.f2) nd7 (q, e, qb)
    [0] = St1
        St1 <- (hardrive.h1.f1) nd7 (q, e, qb)

```

In the following example, the design was elaborated using the default access level (no read or write access to simulation objects). Notice the difference in output between this example and the previous example, where the design was elaborated with full access (`xmelab -access +r+w`). In this example, only the drivers for wires and registers with read access are shown.

```

xcelium> scope -drivers h1
q.....output [3:0]
    q[3] (wire/tri) = St0
        St0 <- (hardrive.h1.f4) nd7 (q, e, qb)
    q[2] (wire/tri) = St0
        St0 <- (hardrive.h1.f3) nd7 (q, e, qb)
    q[1] (wire/tri) = St0
        St0 <- (hardrive.h1.f2) nd7 (q, e, qb)
    q[0] (wire/tri) = St1
        St1 <- (hardrive.h1.f1) nd7 (q, e, qb)

```

The following example lists the drivers for a mixed bus.

```

xcelium> scope -drivers
d.....wire (wire/tri) = StX
No drivers
e.....wire (electrical) = Inf
No drivers
w.....wire [0:2]

```

```
w[1] (wire/tri) = StX
No drivers
w.....wire [0:2]
w[2] (wire/tri) = StX
No drivers
```

The following example lists the source for the current debug scope.

```
xcelium> scope -list
```

The following example lists the source for scope `u1`.

```
xcelium> scope -list u1
```

The following example displays line 12 of the source for the current debug scope.

```
xcelium> scope -list 12
```

The following example lists lines 10 through 15 of the source for the current debug scope.

```
xcelium> scope -list 10 15
```

The following command lists lines from the top of the module through line 10 of the source for the current debug scope.

```
xcelium> scope -list - 10
```

The following command lists lines of source for the current debug scope, beginning with line 30.

```
xcelium> scope -list 30 -
```

The following command shows the output of the `scope -describe` command when you run in regression mode and some objects do not have read or write access.

```
xcelium> scope -describe h1
clk.....input (-RW)
clrb.....input (-RW)
d.....input [3:0]
    d[3] (-RW)
    d[2] (-RW)
    d[1] (-RW)
    d[0] (-RW)
q.....output [3:0]
    q[3] (wire/tri) = St0
    q[2] (wire/tri) = St0
    q[1] (wire/tri) = St0
    q[0] (wire/tri) = St1
f1.....instance of module flop
f2.....instance of module flop
f3.....instance of module flop
```

```
f4.....instance of module flop
```

Related Topic

- [Enabling Read, Write, or Connectivity Access to Digital Simulation Objects](#)

set_analog_param

Specifies the value for the given analog analysis control parameters for the subsequent analog time step. If the specified parameter has an active parameter sweep, the `set_analog_param` command is ignored.

If the command is set multiple times for the same parameter, the latest command takes effect. However, if a new value is specified for the same parameter in the middle of command execution, the parameter value is evaluated according to the newly specified value from the exact conditions it was interrupted at.

Example:

If the target value set for a parameter `temp` is 100° C over 50ns and the command execution is interrupted at `t=30ns`, the slope and ramp time will be calculated from `t=30ns` with the value of `temp` at 30ns.

This command is available only when the analog engine is enabled and does not have an active setup.

Interaction with `$cds_set_temperature`

if `$cds_set_temperature` is set when the `set_analog_param` command is specified for the parameter `temp`, the value of `$cds_set_temperature` overrides the value set through the `set_analog_param` command.

Example:

```
set_analog_param -name temp -value 100 -delay 100ns -nstep 2
```

If the above command is set for 0° C temperature at `t=0ns`, without specifying the value for `$cds_set_temperature`, the following results are displayed.

```
(0ns: 0 Celsius, 50ns: 50 Celsius, 100ns: 100 Celsius)
```

On setting `$cds_set_temperature` to 80 at `t=40ns`, along with this command, the results get updated as follows.

```
(0ns: 0 Celsius, 40ns: 80 Celsius, 50ns: 50 Celsius, 100ns: 100 Celsius)
```

set_analog_param Command Syntax

```
set_analog_param
  -name paramName
  -show
  -value paramValue
  -delay time
  -nstep steps
```

set_analog_param Command Options

This section describes the options that you can use with the Tcl `set_analog_param` command.

<code>-name paramName</code>	Specifies the name of the parameter you want to set the value for. Valid values are <code>steppreset</code>, <code>vabstol</code>, <code>iabstol</code>, <code>reltol</code>, and <code>temp</code>. Example: <pre>set_analog_param -name iabstol</pre>
<code>-show</code> (optional)	Displays the list of available parameters that can be modified along with their current values. Example: <pre>set_analog_param -show steppreset = cx vabstol = 1.000000e-06 iabstol = 1.000000e-12 reltol = 1.000000e-03 temp = 27.0000001</pre>
<code>-value paramValue</code>	Specifies the value for the given parameter and indicates its target value. For real values, either use the scientific format or double format. Example: Sets the value for the parameter <code>temp</code> to 30. <pre>set_analog_param -name temp -value 30</pre>
<code>-delay time</code> (optional)	Specifies the time in which the given parameter must reach its target value. If not specified, the parameter reaches its target value at the next analog time point. Example: If specified at <code>t=30ns</code>, the value for the parameter <code>temp</code> will be updated at <code>t=40ns</code>. <pre>set_analog_param -name temp -value 30 -delay 10ns</pre>

`-nstep steps`

(optional)

Specifies the number of steps required to reach the target value for a parameter.

If not specified, the default number of steps is one.

If the `-delay` option is not specified, the `-nstep` option is ignored.

Example:

If specified at `t=30ns`, the target value (57 Celsius) of the parameter `temp` is reached over ten steps within 10ns, until 40ns is reached.

```
set_analog_param -name temp -value 57 -delay 10ns -nstep 10
```

Related Topic

- [Appendix B](#)

status

The Tcl `status` command displays memory and CPU usage statistics and shows the current simulation time. When the analog solver is active, the delta cycle count is not displayed.

status Command Syntax

```
status
```

status Command Options

None.

status Command Example

The following example shows the type of statistics displayed by the `status` command.

```
xcelium> status
Memory Usage - 8.7M program + 10.8M data = 19.5M total
CPU Usage - 0.1s system + 0.3s user = 0.5s total (0.4% cpu)
Simulation Time - 856 NS + 0
```

stop

Creates or operates on a breakpoint. You can

- Create various kinds of breakpoints (using the `-create` modifier followed by an option that specifies the breakpoint type)
- Display information on breakpoints (`-show`)
- Disable a breakpoint (`-disable`)
- Enable a previously disabled breakpoint (`-enable`)
- Delete a breakpoint (`-delete`)

See the "Setting Breakpoints" section of the "Debugging Your Design" chapter in *Cadence Verilog Simulation User Guide* for more information:

Syntax

```

stop

    -create
        -assert [{-all | -depth {levels | all | to_cells}}] [scope_name]
        -condition {tcl_expression}
        -delta delta_cycle_number [-relative|-absolute]
            [-start delta_cycle_number]
            [-modulo delta_cycle_number]
            [-timestep]
        -iso_rule rule_name [rule_name ...] [-iso_enable | -iso_disable]
        -label "stop_message"
        -line line_number
            [-unit unit_name | [ scope_name ] [-all]]
            [-file filename]
        -object object_names
        -at_resfunc <net> -cv_limit <num>

        -pdname power_domain_name [power_domain_name ...]
        [-isolation [-iso_disable | -iso_enable]]
        [-pd_off]
        [-pd_on]
        [-pd_standby]
        [-pd_trans]
        [-retention [-sr_restore | -sr_save]]

        -process process_name

        -pwr_mode_transition mode_transition_name [mode_transition_name ...]
        -randomize [-always] [object_name ]
        -sr_rule rule_name [rule_name ...] [-sr_save | -sr_restore]
        -subprogram subprogram_name

        -time time_spec [-relative | -absolute]
            [-start time_spec]
            [-modulo time_spec]

        [-continue]
        [-delbreak count]
        [-execute command]
        [-if { tcl_expression}]
        [-name break_name]
        [-silent]
        [-skip count]

    -delete { break_name | pattern} ...

```

```
-disable { break_name | pattern} ...
-enable { break_name | pattern} ...
-show [{ break_name | pattern} ...]
```

The argument to `-delete`, `-disable`, `-enable`, or `-show` can be

- A break name
- A list of break names
- A pattern
 - The asterisk (*) matches any number of characters
 - The question mark (?) matches any one character
 - [characters] matches any one of the characters
- Any combination of literal break names and patterns

Modifiers and Options

Modifiers	Options and Arguments	Function
<code>-create</code>		Creates a breakpoint. This modifier must be followed by an option that specifies the breakpoint type: <pre>-condition -delta (VHDL only) -line -object -process (VHDL only) -time</pre>

```
-assert [{-all | -
depth {levels | all
| to_cells}}]
[scope_name]
```

The `-assert` option lets you define a single breakpoint that is shared by multiple assertions in the design. Stop at failures for all assertions in the design hierarchy by using the `-all` option.

```
xcelium> stop -assert -all
```

Stop at failures for all assertions in a specified scope (or the current debug scope if no scope is specified).

```
xcelium> stop -assert
```

```
xcelium> stop -assert top
```

```
xcelium> stop -assert top.ul
```

Specify how many scope levels to descend when searching for assertions to stop by using the `-depth` option. The argument to `-depth` can be:

- `Levels` - Descend the specified number of scopes. For example, `-depth 1` means include only the given scope, `-depth 2` means include the given scope and its sub-scopes, and so on. The default is 1.
- `all` - Include all scopes in the hierarchy below the given scope.
- `to_cells` - Include all scopes in the hierarchy below the specified scope(s), but stop at cells (Verilog modules with ``celldefine` or VITAL entities with VITAL Level0 attribute).

By using the `-assert` option, you can avoid having to define a whole set of breakpoints on the assertions using separate `stop -object` commands. The `-assert` option can also be used with other `stop` command options, such as `-execute` and `-continue`.

	<pre>-condition {tcl_expression}</pre>	Sets a breakpoint that triggers when any digital object referenced in <code>tcl_expression</code> changes value (wires, signals, registers, and variables) or is written to (memories) and the expression evaluates to true (non-zero, non-x, non-z). <code>tcl_expression</code> must contain at least one digital object. For RNM nets, you can set condition for <code>`wrealXState</code> and <code>`wrealZStates</code> to trigger a breakpoint when the specified state is true. Although condition breakpoints are not triggered by changes in analog objects, you can include analog objects in the conditional expression, and their values are used when the condition is evaluated (due to a digital object changing value). The simulator does not support stop points on individual bits of registers. If a bit-select of a register appears in the expression, the simulator stops and evaluates the expression when any bit of that register changes value. The same holds true for compressed wires. See Tcl Expressions as Arguments for details on the format of conditional expressions. Objects included in a <code>-condition</code> expression must have read access. An error is printed if the object does not have read access. See Enabling Read, Write, or Connectivity Access to Digital Simulation Objects for details.
	<pre>-continue</pre>	Resumes the simulation after executing the breakpoint. The simulator does not go into interactive mode.
	<pre>-delbreak count</pre>	Deletes the breakpoint after it has triggered <code>count</code> number of times.
	<pre>-delta delta_cycle_num [-absolute]</pre>	Sets a breakpoint that triggers when the simulation delta cycle count reaches the specified delta cycle.

```
[-relative]  
  
[-start  
delta_cycle_num]  
  
[-modulo  
delta_cycle_num]  
  
[-timestep]
```

The delta cycle specification can be absolute or relative (the default). If absolute, the breakpoint is automatically deleted after the delta cycle is reached and the breakpoint triggers. If relative, the delta cycle specification is an interval, and the breakpoint stops the simulation every *n* delta cycles.

Use `-start` to specify the absolute delta cycle at which a repetitive breakpoint is to begin firing. If this cycle is before the current cycle, the first stop occurs at the next cycle at which it would have occurred had the stop been set at the cycle specified with `-start`.

The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute delta cycle of the first stop cycle for a repeating delta cycle stop. This differs from `-start` only when the given cycle is more than one repeat interval in the future. In this case, the first stop occurs at a delta cycle less than or equal to one interval in the future such that a stop eventually occurs at the given cycle. For example, if you set a delta breakpoint to stop the simulation every 10 delta cycles, and specify `-modulo 15`, the simulation stops at delta cycle 5, 15, 25, and so on.

`save -environment` writes this option to the script to restore your delta breakpoint pattern.

The `-timestep` option provides a way to detect infinite loops (due to infinite delta cycles) in the design. This option halts the simulation if the specified number of delta cycles is created at any given simulation time. The simulation halts after the first timestep delta cycle is reached, and the simulation cannot be advanced.

See the "Setting a Delta Breakpoint" section of the "Debugging Your Design" chapter in *Cadence Verilog Simulation User Guide* for

		more information.
	<pre>-at_resfunc <net> - cv_limit <num></pre>	Sets a breakpoint when the resolution function value changes for the target user-defined nettype (UDN). The break occurs in the resolution function for any RNM net within the specified scope. Optionally, you can set the breakpoint on the number of convergence iterations using by the <code>cv_limit</code> option. The <code>cv_limit</code> value must be a number larger than the expected number of delta cycle iterations required to resolve a time point. When in resolution function, no other additional argument needs to be added to the TCL <code>driver</code> or <code>value</code> commands to report the net being resolved. The <code>-if {tcl_expression}</code> option can be added to further qualify the stop behavior. In addition, the <code>-execute {tcl_command}</code> or <code>-execFile {tcl_commandfile}</code> options can be used to augment the stop behavior. For example, to display drivers when the stop occurs, specify: <code>-execute driver</code>
	<code>-iso_rule</code>	Stop when the specified isolation rule becomes enabled or disabled.
	<pre>rule_name [rule_name ...] [-iso_enable -iso_disable]</pre>	You can specify multiple isolation rule names. The <code>-iso_rule</code> option has two sub-options: • <code>-iso_disable</code>—Stop only when the isolation rule becomes disabled. • <code>-iso_enable</code>—Stop only when the isolation rule becomes enabled.

	<code>-label</code> <code>"stop_message"</code>	Specify a label for the breakpoint message. The breakpoint will issue the <code>stop_message</code>. For example: <pre>xcelium> stop -line 19 -file testbench.sv -label "setup phase"</pre> Created stop 1 <pre>xcelium> run 0 FS + 0 setup phase</pre>
	<code>-execute command</code>	Executes the specified Tcl command when the breakpoint is triggered. If the command that you want to execute requires an argument, enclose the command and its argument in curly braces. You also can specify that you want to execute a list of commands. Separate the commands with a semicolon. Tcl, however, displays only the output of the last command.
	<code>-if{tcl_expression}</code>	Sets a condition on the breakpoint. The breakpoint triggers only if the given Tcl Boolean expression evaluates to true (non-zero, non-x, non-z). This option can be used with any breakpoint type. For more information on the format of <code>tcl_expression</code>, see Tcl Expressions as Arguments. For RNM nets, you can set condition for <code>`wrealXState</code> and <code>`wrealZStates</code> to trigger breakpoint when the specified state is true. Objects included in an <code>-if</code> expression must have read access. An error is printed if the object does not have read access. See Access to Digital Simulation Objects.

	<pre>-line line_number {-unit unit_name [scope_name] [-all] } [-file filename]</pre>	Sets a breakpoint that triggers when the specified line number is about to execute. You can set breakpoints on both analog and digital code statements. You cannot set a line breakpoint when you are using the simulation front end (SFE) parser. In addition, because structural code is not sequential, you cannot set line breakpoints in such code. You must specify which design unit contains the line. There are two ways to do this: Use <code>-unit</code>. The stop occurs whenever the line number in the specified design unit is about to execute, no matter where in the design hierarchy that unit appears. Specify the name of a particular scope in the design hierarchy. This creates an instance-specific breakpoint. The breakpoint occurs only for that particular instance of the corresponding design unit, no matter where else it may appear in the design hierarchy. To create a breakpoint that is not instance-specific using the <code>scope_name</code> method, use the <code>-all</code> option. If the scope name is omitted, then the current debug scope is used.
		The <code>-file</code> option specifies which of the source files that make up the specified design unit contains the specified line. This is necessary if the design unit has multiple source files. You must compile with the <code>-linedebug</code> option to enable the setting of line breakpoints. See the "Setting a Source Code Line Breakpoint" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.

	<code>-name break_name</code>	Specifies a name for the breakpoint. This name can then be used to delete, disable, or enable the breakpoint. If you do not use <code>-name</code> , breakpoints are numbered sequentially.
	<code>-object object_name</code>	Sets a breakpoint that triggers when the specified object changes value (wires, signals, registers, and variables) or is written to (memories). Note: You cannot create object breakpoints for analog objects. The object specified as the argument must have read access for the breakpoint to be created. An error is printed if the object does not have read access. See Access to Digital Simulation Objects. See the "Setting an Object Breakpoint" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.
<code>-pdname</code>		Set a breakpoint that triggers when the specified power domain changes status. If no options are specified, simulation stops when the power domain is powered down or powered up. You can specify multiple power domain names.
	<pre>power_domain_name [power_domain_name ...] [-isolation [- iso_disable - iso_enable]] [-pd_off] [-pd_on] [-pd_standby] [-pd_trans] [-retention [- sr_restore - sr_save]]</pre>	The <code>-pdname</code> option has several sub-options that let you create power domain breakpoints that trigger under specific conditions: • <code>-pd_off</code>—Stop when the power domain turns off. • <code>-pd_on</code>—Stop when the power domain turns on. This option stops the simulation when the specified power domain has transitioned to the ON state or to the UNINITIALIZED state.

The state of the power domain is UNINITIALIZED when the simulation is being controlled by active state conditions, and when the power domain is on but there are no active state conditions enabled to specify the nominal condition to which the domain should transition.

- `-pd_standby`—Stop when the power domain enters standby mode.
- `-pd_trans`—Stop when the power domain transitions. The breakpoint triggers when the specified power domain starts transitioning from one nominal condition to a different nominal condition.
- `-isolation`—Stop when any isolation rule associated with the power domain is enabled or disabled.
- `-iso_disable`—Stop when any isolation rule associated with the power domain is disabled.
- `-iso_enable`—Stop when any isolation rule associated with the power domain is enabled.
- `-retention`—Stop when any state retention rule associated with the power domain saves or restores its variables.
- `-sr_restore`—Stop when any state retention rule associated with the power domain restores its variables.
- `-sr_save`—Stop when any state retention rule associated with the power domain saves its variables.

	<code>-process process_name</code>	Sets a breakpoint that triggers when the specified VHDL named process starts executing or when it resumes executing after a wait statement. You must compile with <code>-linedebug</code> to enable the setting of process breakpoints. See the "Setting a Process Breakpoint" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.
	<code>-silent</code>	Suppresses the display of the message that is printed when a breakpoint triggers.
	<code>-skip count</code>	Tells the simulator to ignore the breakpoint for the first <code>count</code> times that it triggers. You can use <code>-skip</code> to set a breakpoint on the n^{th} occurrence of an event; in particular, you can use it to get inside <code>for</code> loops.
<code>- pwr_mode_transition</code>	<code>mode_transition_name mode_transition_name ...]</code>	Stop when the specified power mode transition starts and ends.
<code>-randomize</code>		Sets a breakpoint in SystemVerilog <code>randomize()</code> method calls. The SystemVerilog built-in <code>randomize()</code> function returns the value 1 for success or 0 for failure. A failure occurs because there are conflicts in the collection of constraints to solve or because a variable is over-constrained.

	<pre>-always] [object_name]</pre>	The <code>stop -create -randomize</code> command lets you set a breakpoint in <code>randomize()</code> method calls. You can then use other Tcl commands, such as <code>deposit -constraint_mode</code>, <code>deposit -rand_mode</code>, <code>constraint</code>, and <code>run -rand_solve</code>, to debug the randomization failures. By default, simulation stops at the end of <code>randomize()</code> calls when the call is about to return 0, or failure. If you include the <code>-always</code> option, simulation stops for all <code>randomize()</code> calls, regardless of the return status of the call. You can include an <code>object_name</code> argument to stop the simulation in specific <code>randomize()</code> calls. The argument can be a class name or a module name. The simulator stops on a failure in any call of the <code>randomize()</code> method in the specified module or with the specified class name. If <code>-always</code> is specified, the simulator stops in all calls to the <code>randomize()</code> method. <code>stop -randomize</code> commands are supported for calls to class and scope <code>randomize</code> methods.
<code>-sr_rule</code>		Stop when the specified state retention rule saves or restores its variables.
	<pre>rule_name [rule_name ...] [-sr_save - sr_restore]</pre>	You can specify multiple state retention rule names. The <code>-sr_rule</code> option has two sub-options: • <code>-sr_restore</code>: Stop only when the state retention rule restores its variables. • <code>-sr_save</code>: Stop only when the state retention rule saves its variables.

<code>-subprogram</code>		Set a breakpoint that triggers when the specified VHDL subprogram or Verilog task or function is called.
	<code>subprogram_name</code>	You must compile with the <code>-linedebug</code> option to enable the setting of subprogram breakpoints.

	<pre>-time time_spec [-absolute] [-relative] [-start time_spec] [-modulo time_spec]</pre>	Sets a breakpoint that triggers at the specified time. The time can be absolute or relative (the default). Absolute time breakpoints are automatically deleted after they trigger. Relative time breakpoints are periodic, stopping, for example, every 10 ns. The digital solver is always active when the simulator stops for a time breakpoint. Use <code>-start</code> to specify the absolute simulation time at which a relative time breakpoint is to begin firing. If this time is before the current simulation time, the first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with <code>-start</code>. The <code>-modulo</code> option is similar to <code>-start</code>. Use <code>-modulo</code> to specify the absolute simulation time of the first stop time for a repeating stop. This differs from <code>-start</code> only when the given time is more than one repeat interval in the future. In this case, the first stop occurs at a time less than or equal to one interval in the future such that a stop eventually occurs at the given time. For example, if you set a time breakpoint to stop the simulation every 100 ns, and specify <code>-modulo 250</code>, the simulation stops at time 50, 150, 250, and so on. When you execute a <code>save -environment</code> command to save your debug environment, this option is written to the script to restore your time breakpoint pattern. See the "Setting a Time Breakpoint" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.

-disable	{break_name pattern} ...	Disables the breakpoints specified by the argument without deleting them. See the "Disabling, Enabling, Deleting, and Displaying Breakpoints" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.
-enable	{break_name pattern} ...	Enables the previously disabled breakpoints specified by the argument. See the "Disabling, Enabling, Deleting, and Displaying Breakpoints" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.
-delete	{break_name pattern} ...	Deletes the breakpoints specified by the argument. See the "Disabling, Enabling, Deleting, and Displaying Breakpoints" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.
-show	[{break_name pattern} ...]	Shows the status of the breakpoints specified by the argument. If no breakpoint is specified, all breakpoints are shown. See the "Disabling, Enabling, Deleting, and Displaying Breakpoints" section of the "Debugging Your Design" chapter in <i>Cadence Verilog Simulation User Guide</i> for more information.

Example

Object Breakpoints

The following command creates a breakpoint that stops simulation when `sum` changes value. The `-create` modifier is not required. Because the `-name` option is not included to specify a breakpoint name, `xmsim` assigns a sequential number as the name. This breakpoint is called `1`.

```
xcelium> stop -create -object sum  
Created stop 1
```

The following command creates a breakpoint named `mybreak` that stops simulation when `sum` changes value.

```
xcelium> stop -object sum -name mybreak  
Created stop mybreak
```

The following command creates a breakpoint that stops simulation when `EEnet_net1` changes value.

```
xcelium> stop -object EEnet_net1  
Created stop 1  
xcelium> run 5 ns  
0 FS + 1 (stop 1: top.EEnet_net1 = '{V:4.9505, I:0, R:99.0099})
```

The following command creates a breakpoint that stops simulation when resolution function value changes. `net1` is a user defined nettype that has a user defined resolution function.

```
xcelium> stop -create -at_resfunc top.net1  
Created stop 1  
xcelium> run  
  
50 NS + 1 (stop 1: Resolution Function udt_package::Tavg)  
resolving net: top.net1
```

The following command creates a breakpoint that triggers when `sum` changes value. The breakpoint is ignored the first 3 times it triggers.

```
xcelium> stop -object sum -skip 3
```

The following command creates a breakpoint that stops simulation when `clr` changes value.

The `value data` command is executed when the breakpoint triggers. Because the `value` command requires an argument, it must be enclosed in curly braces.

```
xcelium> stop -object clr -execute {value data}
```

The following command creates a breakpoint that triggers when `clr` changes value. The `value data` command is executed when the breakpoint triggers. The `-continue` option prevents the simulator from entering interactive mode every time the stop triggers.

```
xcelium> stop -object clr -execute {value data} -continue
```

The following command creates an object breakpoint that triggers when `data` changes value. The `-delbreak` option specifies that the breakpoint is deleted after it triggers three times.

```
xcelium> stop -object data -continue -delbreak 3
```

The following command creates a breakpoint that triggers when `clk` changes value, but only if `clk` is

high. See [Tcl Expressions as Arguments](#) for details on the syntax of the argument to the `-if` option.

```
xcelium> stop -object clk -if {#clk == 1} -continue
```

The following command creates a breakpoint that triggers when `data[1]` has the value `1` and the time becomes greater than 3 ns.

```
stop -object data -if {#data[1] == 1 && [time ns -nount] > 3}
```

The following command shows the error message that is displayed if you run in regression mode and then try to set an object breakpoint on an object that does not have read access.

```
xcelium> stop -object clk|
xmsim: *E,RDACRQ: Object does not have read access: hardrive.clk.
```

The following shows an error caused by trying to create a breakpoint on an analog object.

```
xcelium> stop -object compSig
xcelium: *W,STALOB: Cannot set stop on analog object:
    top.dac.compSig. This object ignored.
xmsim: *E,STOBEX: Object expected after -OBJECT
    option of stop command.
```

Line Breakpoints

The following command creates a breakpoint that stops simulation when line number 10 in the current debug scope is about to execute.

```
xcelium> stop -line 10
```

The following command creates a breakpoint that stops simulation when line number 13 in scope `counter` is about to execute.

```
xcelium> stop -line 13 counter
```

In the following command, the `-all` option specifies that the stop is noninstance-specific. The breakpoint occurs on all scopes which are instances of the same module. For example if there are two instances of module `m16`, as follows:

```
module board;
<declarations>m16 counter1 (...);
m16 counter2 (...);
<code>
endmodule
```

the breakpoint triggers when line 13 in either `counter1` or `counter2` is about to execute.

```
xcelium> stop -line 13 counter1 -all
```

The following command is equivalent to the command shown in the previous example. Both commands create non-instance-specific breakpoints.

```
xcelium> stop -line 13 -unit m16
```

In the following example, the `-file` option specifies which of the source files that make up the given scope (or the debug scope if none is given) contains the specified line. This is necessary if the scope has multiple source files.

```
xcelium> stop -line 13 counter -file foo.v
```

Time Breakpoints

The following command creates a breakpoint that stops simulation at absolute time 200 ns. The breakpoint is automatically deleted after it triggers.

```
xcelium> stop -time 200 ns -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns and then executes the `value` command. The `-relative` option is the default for time breakpoints.

```
xcelium> stop -time 200 ns -relative -execute {value data}
```

The following command creates a repetitive breakpoint that stops the simulation every 200 ns. The `-start` option specifies the absolute time at which the breakpoint starts. For example, if the current simulation time is 300 ns, the breakpoint stops the simulation at time 600, 800, 1000, and so on.

```
xcelium> stop -time 200 ns -start 600 ns
```

In the following example, the current simulation time is 300 ns. The absolute time specified with `-start` is before the current simulation time. The first stop occurs at the next future time at which it would have occurred had the stop been set at the time specified with `-start`. In this example, the first stop occurs at time 450 ns.

```
xcelium> stop -time 200 ns -start 250 ns
```

The following example shows how the `-modulo` option is used to save a breakpoint pattern. Suppose that you simulate to time 300 ns and then set a repetitive breakpoint with the following command:

```
xcelium> stop -time 200 ns -start 350 ns
```

This command stops the simulation at time 350, 550, 750, and so on. If you then execute a `save -environment` command to save your debug environment, the following line is written to the script:

```
stop -create -name 1 -time 200 NS -relative -modulo 950 NS
```

If you then exit and re-enter the simulation and source the script containing this command, the breakpoint pattern is re-established. In this example, if you restart the simulation and start at time 0, the

breakpoint triggers the first time at time 150. It then triggers at 350, 550, 750, and so on.

The following command includes the `-if` option to set a breakpoint at time 100 ns (relative) if `data[1]` has the value 1.

```
xcelium> stop -time 100 ns -if {#data[1] == 1}
```

Delta Breakpoints

The following command creates a breakpoint that stops the simulation when it reaches 20 delta cycles. The breakpoint is automatically deleted after it triggers.

```
xcelium> stop -delta 20 -absolute
```

The following command creates a repetitive breakpoint that stops the simulation every 10 delta cycles. The `-start` option specifies the absolute delta cycle at which the breakpoint starts. For example, if the current delta cycle count is 0, the breakpoint stops the simulation when the delta cycle count is 30, 40, 50, and so on.

```
xcelium> stop -delta 10 -start 30
```

Condition Breakpoints

In a condition breakpoint, the argument to the `-condition` option is a Tcl expression. See [Tcl Expressions as Arguments](#).

The following command sets a condition breakpoint that stops the simulation when `count`, the output of a 32-bit counter, has the value 100, decimal. The signal `count` is available from the top level of the hierarchy.

```
Verilog: xcelium> stop -condition {[value %d top.count] = 100}  
VHDL: xcelium> stop -condition {[value %d :count] = 100}
```

If you are currently at the top level, you can omit the hierarchical path specification to `count`, and the two commands shown in the previous example could be written as follows:

```
xcelium> stop -condition {[value %d count] = 100}
```

The `value` command uses the value of the `vlog_format` (or `vhdl_format`) variable. If you set the value of this variable to `%d`, the command shown in the previous example could be written as follows:

```
xcelium> stop -condition {[value count] = 100}
```

Instead of using the `value` command to get the value of `count` into the expression evaluator, you can use `#count`. Include the format specifier after the `#` sign.

For Verilog, you can use the standard notation (for example `4'b0011`). For example, you can set the breakpoint on `count` as follows:

VHDL does not have the same type of notation. Vectors must be enclosed in quotation marks, as shown in the next example.

The following command sets a condition breakpoint that stops the simulation when bit 0 of `count` is 1. The expression is evaluated when any bit of `count` changes value. For VHDL, single-bit entities must be enclosed in single quotation marks.

The following command is identical to the previous command. An explicit `value` command is used to get the value of `count` (bit 0) into the expression parser.

In the following command, the `-if` option is used to conditionalize the condition breakpoint. This breakpoint stops the simulation at the next positive edge of the clock if `en1` or `en2` is 1.

The following command stops the simulation at 5 ns (absolute time). After that, `clock` changes depending on the condition in the `if` expression, and this happens repeatedly every 5 ns. The `continue` option is used to prevent the simulation from stopping every time the breakpoint triggers. VHDL requires use of the single quotation marks.

Process Breakpoints

The following command sets a breakpoint that stops the simulation whenever the process called `:load_action` is executed.

```
xcelium> stop -process :load_action
```

Examples of Other stop Command Modifiers

The following command sequence illustrates the `-show` modifier. The first command creates a source line breakpoint called `break1`; the second creates an object breakpoint called `break2`. The third command shows the status of the two breakpoints.

```
xcelium> stop -line 12 -name break1
Created stop break1
xcelium> stop -object data -name break2
Created stop break2
xcelium> stop -show
break1 Enabled Line: ./shortdrive.v:12 (scope: top)
break2 Enabled Object top.data
xcelium>
```

In the following command sequence, breakpoint `break1` is first disabled with the `-disable` modifier and then enabled with the `-enable` modifier.

```
xcelium> stop -show
break1 Enabled Line: ./shortdrive.v:12 (scope: top)
break2 Enabled Object top.data
xcelium> stop -disable break1
xcelium> stop -show
break1 Disabled Line: ./shortdrive.v:12 (scope: top)
break2 Enabled Object top.data
xcelium> stop -enable break1
xcelium>
```

The following command deletes breakpoint `break1`.

```
xcelium> stop -delete break1
```

To disable, enable, or delete the two breakpoints `break1` and `break2`, any of the following commands could be used.

```
xcelium> stop -delete *1 *2
xcelium> stop -delete break?
xcelium> stop -delete br*
```

The following command displays information on any breakpoint beginning with `v` or `b`.

```
xcelium> stop -show {[vb]*}
```

Tcl Expressions as Arguments

The `stop` command has two options that let you specify conditions. Both options require a Tcl expression argument.

- `-condition`

This option specifies that you are creating a condition breakpoint, as opposed to some other kind of breakpoint, such as a time or object breakpoint. A condition breakpoint triggers when any digital object named in the Tcl expression has an event that would trigger an object breakpoint and the expression evaluates to non-zero, non-x, or non-z. Although condition breakpoints are not triggered by changes in analog objects, you can include analog objects in the conditional expression and their values are used when the condition is evaluated (due to a digital object changing value).

- `-if`

This option can be used with any breakpoint type, including condition breakpoints. The Tcl expression argument is evaluated, and the stop triggers if the expression evaluates to non-zero, non-x, or non-z.

There are two general rules to keep in mind when writing the Tcl expression:

- Enclose the expression in braces to suppress immediate substitution of values.

`{tcl_expression}`

If you are using the SimVision environment, these braces are included on the Set Break form.

In the following example, the value of `w[1]` would be substituted with its current value (1'b~~x~~, for example) if there were no braces. No object would be named in the expression by the time the `stop` command routine sees it, resulting in an error.

```
xcelium> stop -condition #w[1] == 1
xcelium> stop -condition {#w[1] == 1}
```

- You must use either an explicit `value` command or the `#` character to get the object's value into the expression parser because the parser does not understand names. For example, the following command generates an error message.

```
xcelium> stop -time 100 ns -if {r[1] == 1}
```

Use the following commands:

Verilog:

```
xcelium> stop -time 100 ns -if {[value r[1]] == 1'b1}
xcelium> stop -time 100 ns -if {#r[1] == 1}
```


VHDL:

```
xcelium> stop -time 100 ns -if {[value r(1)] == `1'}  
xcelium> stop -time 100 ns -if {#r(1) == `1'}
```

Format specifiers can be used with either the `value` command or the `#` sign. If you use the `#` sign, place the format specifier after the `#` sign. For example,

Verilog:

```
xcelium> stop -condition {[value %d out] = 12}  
xcelium> stop -condition {#%dout = 12}
```

VHDL:

```
xcelium> stop -condition {[value %d out] = 12}  
xcelium> stop -condition {#%dout = 1}
```

For VHDL, you must enclose vectors in quotation marks and single-bit entities in single quotation marks. For example,

```
Verilog: xcelium> stop -condition {#clock == 1}  
VHDL: xcelium> stop -condition {#clock == `1'}  
  
Verilog: xcelium> stop -condition {#count = 4'b0101}  
VHDL: xcelium> stop -condition {#clock = "0101"}
```

See the "Basics of Tcl" appendix in *Cadence Verilog Simulation User Guide* for more details on basic Tcl syntax and on the extensions to Tcl that have been added to handle types and operators of the Verilog and VHDL hardware description languages.

For RNM nets, the following command creates a conditional on-change break when `top.rnmnet` changes to ``wrealXState`:

```
stop -create -object -if {#top.rnmnet = "`wrealXState"} top.rnmnet
```

strobe

The Tcl `strobe` command prints the values of specified VHDL, Verilog, or SystemC objects based on one of the following specifications:

- when a specified condition is true
- when a specified signal changes value
- at a specified time interval

The `strobe` command is a Tcl procedure that uses the `stop` command to set a condition, object, or time breakpoint and then, when the breakpoint triggers, executes a `value` command to print out the values

of the specified objects in tabular format.

You can apply the `strobe` command to objects in the SystemC portion of the design that are derived from `sc_signal`, `sc_port`, `sc_export`, `sc_in`, `sc_inout`, `sc_clock`, and `ncsc_viewable`. You can only set one strobe at a time. Setting a new strobe automatically deletes the previous strobe.

By default, the simulator prints the values of the specified objects using the default format of the `value` command. You can specify a format for individual objects by enclosing the object-format pair in curly braces. For example:

```
xcelium> strobe -time 100 clk {data %b}
```

You can also set the `strobeFmt` variable to specify a global format. For example:

```
xcelium> set strobeFmt %b
```

If you interrupt or stop the simulation (with `CTRL/C`, an `assert` statement, or by running the simulation for a specified period of time, for example), and then continue the simulation, the simulator does not print the header in the tabular output. This is done so that if you send the output to a file with the `-redirect` option, the header does not appear in the output file every time you continue the simulation. However, if you are sending output to the screen, you might want to redisplay the header. To display the header again, set the `strobeHeader` variable to `1`, as follows:

```
xcelium> set strobeHeader 1
```

The variable `strobeTimeWidth` can be used to control how much space is used to print the simulation time in the strobe output. By default, the value is `15`. To change this, set the `strobeTimeWidth` variable before creating the strobe. For example:

```
xcelium> set strobeTimeWidth 25
```

The `strobe` command defines other Tcl variables that are internal to the operation of the command.

The following variables should not be changed:

- `strobeObjects`
- `strobeObjectList`
- `strobeStream`
- `headerList`
- `widthList`

If a strobe is already set, you must reset the strobe to the change to take effect.

strobe Command Syntax

```
strobe <strobe_specification> <object_list> [output_file_specification]
      -delete
      -help
```

strobe Command Options

This section describes the options that you can use with the Tcl `strobe` command.

`<strobe_specification>`

Specifies a condition for the strobe. Signal values are displayed when a specified condition is true (`-condition`), when a specified signal changes value (`-object`), or at a specified time interval (`-time`).

When using this command you must choose one of the three available methods to display signal values for simulation objects:

- `-condition <condition_specification>`:
Displays the values of the specified object(s) when the `condition_specification` is true.

Set a breakpoint that triggers when any object referenced in the `tcl_expression` changes value (wires, signals, registers, and variables) or is written to (memories) and the expression evaluates to true (non-zero, non-x, non-z).
- `-object <object_specification>`: Displays the values of the specified object(s) when the object specified by the `object_specification` argument changes value.

With `-object`, you can specify only one object to monitor for a change in value. For example, the following command displays the values of `y` and `z` when `x` changes value.


```
xcelium> strobe -object x y z
```
- `-time <time_specification>`: Displays the values of the object(s) at the time interval specified by the `time_specification` argument.


```
xcelium> strobe -time 100ns x y z
xcelium> strobe -time 100ns x {y %b} z
```

`object_list` *object*

Refers to the signals the strobe is being applied to and the format is basically how to display the object such as in hex format and binary format.

`output_file_specification`
`-redirect` *filename*
`[-append]`

Saves output to the specified file. By default, the simulator prints the output of the `strobe` command to the screen.

The `-redirect` option redirects output to the specified file. By default, the simulator prints the output of the `strobe` command to the screen.

Include the `-append` option to append output to the file specified with the `-redirect` option.

`-delete`

Deletes the strobe.

`-help`

Displays help on the `strobe` command. You can also get help on the `strobe` command by using the `help` command.

strobe Command Examples

The following command prints the values of signals `clk`, `clr`, `data`, and `q` every 100 ns.

```
xcelium> strobe -time 100 clk clr data q
Setting up strobe time - '100'
xcelium> run
Time |clk |clr |data |q |
-----
100 NS |1'h1 |1'h1 |4'h0 |4'hx |
200 NS |1'h1 |1'h1 |4'h1 |4'h0 |
300 NS |1'h1 |1'h1 |4'h2 |4'h1 |
400 NS |1'h1 |1'h1 |4'h3 |4'h2 |
...
```

The following command prints the values of signals `clk`, `clr`, `data` (in binary), and `q` every 100 ns.

```
xcelium> strobe -time 100ns clk clr {data %b} q
Setting up strobe time - '100ns'
xcelium> run 300 ns
Time |clk |clr |data |q |
-----
100 NS |1'h1 |1'h1 |4'b0000 |4'hx |
200 NS |1'h1 |1'h1 |4'b0001 |4'h0 |
300 NS |1'h1 |1'h1 |4'b0010 |4'h1 |
Ran until 300 NS + 0
```

In the following command sequence, the simulation is run for 300 ns. When the simulation is resumed, the strobe header is not displayed. The `strobeHeader` variable is then set to display the header when the simulation is resumed.

```
xcelium> strobe -time 100 clk clr data q
```

```
Setting up strobe time - '100'
```

```
xcelium> run 300
```

```
Time |clk |clr |data |q |
```

```
-----
```

```
100 NS |1'h1 |1'h1 |4'h0 |4'hx |
```

```
200 NS |1'h1 |1'h1 |4'h1 |4'h0 |
```

```
300 NS |1'h1 |1'h1 |4'h2 |4'h1 |
```

```
Ran until 300 NS + 0
```

```
xcelium> run 300
```

```
400 NS |1'h1 |1'h1 |4'h3 |4'h2 |
```

```
500 NS |1'h1 |1'h1 |4'h4 |4'h3 |
```

```
600 NS |1'h1 |1'h1 |4'h5 |4'h4 |
```

```
Ran until 600 NS + 0
```

```
xcelium> set strobeHeader 1
```

```
1
```

```
xcelium> run 300
```

```
Time |clk |clr |data |q |
```

```
-----
```

```
700 NS |1'h1 |1'h1 |4'h6 |4'h5 |
```

```
800 NS |1'h1 |1'h1 |4'h7 |4'h6 |
```

```
900 NS |1'h1 |1'h1 |4'h8 |4'h7 |
```

```
Ran until 900 NS + 0
```

To change the format globally, you can set the `strobeFmt` variable, as shown in the following example. You must reset the strobe after setting the `strobeFmt` variable.

```
xcelium> strobe -time 100 clk clr data q
```

```
Setting up strobe time - '100'
```

```
xcelium> run 300
```

```
Time |clk |clr |data |q |
```

```
-----
```

```
100 NS |1'h1 |1'h1 |4'h0 |4'hx |
```

```
200 NS |1'h1 |1'h1 |4'h1 |4'h0 |
```

```
300 NS |1'h1 |1'h1 |4'h2 |4'h1 |
```

```
Ran until 300 NS + 0
```

```
xcelium> set strobeFmt %b
```

```
%b
```

```
xcelium> strobe -time 100 clk clr data q
```

```
Setting up strobe time - '100'
```

```
xcelium> run 300
```

```
Time |clk |clr |data |q |
```

```
-----  
400 NS |1'b1 |1'b1 |4'b0011 |4'b0010 |  
500 NS |1'b1 |1'b1 |4'b0100 |4'b0011 |  
600 NS |1'b1 |1'b1 |4'b0101 |4'b0100 |  
Ran until 600 NS + 0
```

The following command displays the values of `clk`, `clr`, and `q` when data changes value.

```
xcelium> strobe -object data clk clr q
```

```
Setting up strobe object - 'data'
```

```
xcelium> run 300 ns
```

```
Time |clk |clr |q |
```

```
-----  
0 NS |1'h0 |1'h1 |4'hx |  
100 NS |1'h1 |1'h1 |4'hx |  
200 NS |1'h1 |1'h1 |4'h0 |  
Ran until 300 NS + 0
```

The following command displays the values of `data`, `clk`, `clr`, and `q` when data changes value.

```
xcelium> strobe -object data data clk clr q
```

```
Setting up strobe object - 'data'
```

```
xcelium> run 300 ns
```

```
Time |data |clk |clr |q |
```

```
-----  
0 NS |4'h0 |1'h0 |1'h1 |4'hx |  
100 NS |4'h1 |1'h1 |1'h1 |4'hx |  
200 NS |4'h2 |1'h1 |1'h1 |4'h0 |  
Ran until 300 NS + 0
```

The following command displays value of `UDN_EEnet_net1` when `EEnet_bus[0]` changes value.

```
xcelium> strobe -object EEnet_bus[0] EEnet_net1
```

```
Setting up strobe object - 'EEnet_bus[0]'
```

```
xcelium> run 500 ns
```

```
Time |EEnet_net1 |
```

```
-----  
0 NS |'{V:4.9505, I:0, R:99.0099} |  
100 NS |'{V:0, I:0.05, R:0} |  
150 NS |'{V:3.33333, I:0, R:66.6667} |
```

```
250 NS      | '{V:0, I:0.05, R:0}      |
300 NS      | '{V:3.33333, I:0, R:66.6667}    |
400 NS      | '{V:5, I:0, R:100}      |
450 NS      | '{V:0, I:0.05, R:0}      |
```

The following command displays value of `EEnet_net1 EEnet_bus[0]` when `EEnet_net1` changes value.

```
xcelium> strobe -object EEnet_net1 EEnet_net1 EEnet_bus[0]
Setting up strobe object - 'EEnet_net1'
xcelium> run 500 ns
Time      | EEnet_net1                      | {EEnet_bus[0]}      |
-----
0 NS      | '{V:4.9505, I:0, R:99.0099}    | '{V:4.85437, I:0, R:97.0874}
          |
100 NS    | '{V:0, I:0.05, R:0}           | '{V:`wrealXState, I:0, R:`wrealZState}
          |
150 NS    | '{V:3.33333, I:0, R:66.6667}  | '{V:2, I:0, R:40}   |
250 NS    | '{V:0, I:0.05, R:0}           | '{V:`wrealXState, I:0,
R:`wrealZState}      |
300 NS    | '{V:3.33333, I:0, R:66.6667}  | '{V:2, I:0, R:40}   |
400 NS    | '{V:5, I:0, R:100}           | '{V:5, I:0, R:100   |
450 NS    | '{V:0, I:0.05, R:0}           | '{V:`wrealXState, I:0, R:`wrealZState}
          |
```

The following command displays the values of `data` (in binary), `clk`, `clr`, and `q` (in hex) when `data` changes value. Notice that the `object-format` pair is enclosed in curly braces.

```
xcelium> strobe -object data {data %b} clk clr q
Setting up strobe object - 'data'
xcelium> run 300 ns
Time |data|clk|clr|q|
-----
0 NS |4'b0000|1'h0|1'h1|4'hx|
100 NS |4'b0001|1'h1|1'h1|4'hx|
200 NS |4'b0010|1'h1|1'h1|4'h0|
Ran until 300 NS + 0
```

The following command displays the values of the specified signals when `data` has the value 3, decimal. The signal `data` is available from the top level of the hierarchy.

Verilog:

```
xcelium> strobe -condition {[value %d top.data] = 3} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {[value %d :data] = 3} clk clr data q
```

If you are currently at the top level, you can omit the hierarchical path specification to data, and write the two commands shown in the previous example as follows:

```
xcelium> strobe -condition {[value %d data] = 3} clk clr data q
```

Instead of using the value command to get the value of data into the expression evaluator, you can use #data. Include the format specifier after the # sign.

```
xcelium> strobe -condition {#%ddata = 3} clk clr data q
```

The following command displays the values of the specified signals when data has the value 0010 (binary). For VHDL, you must enclose vectors in quotation marks.

Verilog:

```
xcelium> strobe -condition {#data = 4'b0010} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {#data = "0010"} clk clr data q
```

The following command displays the values of the specified signals when bit 0 of data is 1. The expression is evaluated when any bit of data changes value. For VHDL, you must enclose singlebit entities in single quotation marks.

Verilog:

```
xcelium> strobe -condition {#data[0] == 1} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {#data(0) == `1'} clk clr data q
```

The following command is identical to the previous command except that it uses an explicit value command to get the value of data (bit 0) into the expression parser.

Verilog:

```
xcelium> strobe -condition {[value %b data[0]] == 1'b1} clk clr data q
```

VHDL:

```
xcelium> strobe -condition {[value %b data(0)] == `1'} clk clr data q
```

The following command displays the values of the specified signals if the value of `top.load` has the value `x`. Notice that in the Tcl expression, the case-equality operator (`===`) is used.

For this operator, bits that are unknown are included in the comparison, and the result of the expression is always 1 (true) or 0 (false).


```
xcelium> strobe -condition {#top.load === 1'bx} clk clr data q
```

In the following command, the logical comparison operator (= or ==) is used. These operators return the unknown value (x) if either operand is unknown. In a conditional expression, an unknown result is treated as false. Therefore, the following command does not stop the simulation when the signal has the value x.

```
xcelium> strobe -condition {#top.load == 1'bx} clk clr data q
```

Related Topic

- [Enabling Read, Write, or Connectivity Access to Simulation Objects](#)

time

The Tcl `time` command displays the current simulation time scaled to the specified unit. The unit can be

- A time unit that you specify
- `auto`: use the largest base unit that makes the numeric part of the time an integer
- `module`: use the timescale of the current debug scope

The simulation time can be displayed in the following time units:

- `fs`: femtoseconds
- `ps`: picoseconds
- `ns`: nanoseconds
- `us`: microseconds
- `ms`: milliseconds
- `sec`: seconds
- `min`: minutes
- `hr`: hours

If no unit is given, the value of the `$display_unit` variable is used. This variable is set to `auto` by default.

time Command Syntax

```
time [[10 | 100] time_unit | auto | module]
    -delta
    -nounit
```

time Command Options

This section describes the options that you can use with the Tcl `time` command.

`-delta`

Includes the delta cycle count.

The `-delta` option is ignored if the analog solver is active.

At any given simulation time, values of nets are first updated and then behaviors that are sensitive to those nets are executed. This two step process may be repeated any number of times because of zero-delays. The delta cycle count represents the number of times the process is repeated for the given simulation time.

`-nounit`

Does not include the time unit.

time Command Examples

```
xmsim -tcl boardxmsim: v1.0.(p2): (c) Copyright 1995 - 2000 Cadence Design Systems
xcelium> run 100 ns5 count= X, f=x, af=xRan until 100 NS + 0
```

The following command displays the current simulation time in ns.

```
xcelium> time ns100 NS
```

The following command displays the current simulation time in fs.

```
xcelium> time fs100000000 FS
```

The following command displays the current simulation time in 100 times the base unit of fs.

```
xcelium> time 100fs1000000 100FS
```

The following commands illustrate the `auto` keyword, which displays the time using the largest base unit that makes the numeric part of the time an integer.

```
xcelium> time fs100000000 FSxcelium> time auto100 NS
```

The following command displays the current simulation time using the timescale of the current debug

scope.

```
xcelium> time module100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope and including the delta cycle count.

```
xcelium> time module -delta100 NS + 0
```

The following command displays the current simulation time with no time unit.

```
xcelium> time -nounit100
```

value

The Tcl `value` command prints the current value of the specified objects using the last format specifier preceding the object name argument. If no format is specified, a default format is used.

Objects specified as arguments to the `value` command must have read access. An error is printed if an object does not have read access.

The `value` command is supported on user-defined nettypes (UDNs).

value Command Syntax

```
value [format ...] object_name...  
    -potential  
    -flow
```

value Command Options

This section describes the options that you can use with the Tcl `value` command.

formats

You can use the following format specifiers to specify the formats:

- `%c`: character
- `%s`: string
- `%b`: binary
- `%d`: unsigned decimal

Use the `-signed` option to print the value as a signed decimal value.

- `%o`: octal
- `%x`: hexadecimal

- %h: same as %x
- %f: floating-point number
- %e: real number in mantissa-exponent form
- %g: use %e or %f, whichever is shorter
- %t: decimal time scaled from the timescale of the object's module to the simulation's timescale
- %v: strength value—wires only

To revert to the default format, use %.

If no format is specified, the default format depends on the object type. The following are the default formats:

- %d: time
- %d: integer
- %g: real
- \$vlog_format87: reg
- \$vlog_format: wire

The \$vlog_format is a predefined Tcl variable that defaults to %h. You can set this variable to %b, %o, or %d.

For VHDL, values are returned in a format that resembles the appropriate VHDL syntax for the object type. If one of the radix format specifiers (%b, %o, %d, or %x) is given, the format affects the format of integer values and of `bit` and `std_logic` values. The value is shown without any added VHDL syntax, such as double or single quotes. Otherwise, the format specifier is ignored for VHDL values. The VHDL literal format (%v) is the default format. For example:

```
xcelium> value :vector
"101010111100"
xcelium> value %b :vector
101010111100
xcelium> value %h :vector
ABC
xcelium> value :enum
'1'
xcelium> value %d :enum
1
xcelium> value %b :enum
1
```

The value of a digital net that is associated with a mixed signal depends on whether the enclosing module is an ordinary module or a connect module. The value of a digital net within an ordinary module is the resolved value of the connect module drivers that drive the net. The value of a digital net within a connect module is the resolved value of the ordinary module drivers that drive the net. See ["Driver-Receiver Segregation"](#).

You can also use the pound sign (#) as a shortcut to the `value` command. When used on an analog branch, the # shortcut accesses the potential across the branch, not the

flow.

You can use wildcard characters in the argument to a `value` command.

- The asterisk (`*`) matches any number of characters.
- The question mark (`?`) matches any one character.

You cannot use wildcard characters inside escaped names

`-potential`

Returns the potential of analog branches that are specified on the command line immediately following the `-potential` option. This option is ignored for any other kind of object.

`-flow`

Returns the flow value of analog branches that are specified on the command line immediately following the `-flow` option. Returns the flow value of analog objects that have existing Tcl current probes. This option is ignored for any other kind of object.

value Command Example

You have an analog branch declared in Verilog-AMS source code like this:

```
branch (p,n) res;
```

You can return the potential of the branch like this:

```
xcelium> value res  
0.626
```

Where the flow of the `res` and `bar` branches are 0.111mA and 2.3mA respectively, and the potential of the `p_n` branch is 4.666V, using the command

```
xcelium> value -flow res bar -potential p_n
```

returns

```
0.000111 0.00023 4.666
```

The following sequence of `value` commands displays the current value of `data` in a variety of formats.

```
xcelium> value data  
4'h2  
xcelium> value %o data  
4'o02  
xcelium> value %b data  
4'b0010  
xcelium> value %d data  
4'd2  
xcelium> value %g data  
2  
xcelium> value %f data
```

```
2.000000
xcelium> value %e data
2.000000e+00

xcelium> value %b data %d q
4'b0010 4'd1
xcelium> value % data %d data %b data
4'h2 4'd2 4'b0010
```

The following command shows the error message that is displayed when you run in regression mode and use the `value` command on an object that does not have read access.

```
xcelium> value clk
xmsim: *E,RDACRQ: Object does not have read access: hardrive.clk.
```

Related Topics

- [Specifying Unnamed Branch Objects in Tcl Commands](#)
- [User-Defined Nettype and Resolution Function](#)

where

The Tcl `where` command displays the current location of the simulation. This includes the current simulation time and the current scope.

where Command Syntax

```
where
```

where Command Options

None.

where Command Example

```
xcelium> where
TIME: 3400 NS + 0
Scope is (board.counter)
xcelium>
```

Specifying Unnamed Branch Objects in Tcl Commands

Verilog[®]-AMS modules can contain unnamed branches. For example, you might have:

```
electrical a, b;  
V(a,b) <+ 7 * I(a,b) ; // Uses an unnamed branch across a and b
```

To refer to the unnamed branch when you use a Tcl command, you use an underscore between the two nets. So you might have a Tcl command and response like the following:

```
xcelium> describe a_b  
a_b.....analog net (electrical ) = 0
```

Unfortunately, this approach is sometimes ambiguous. For example, consider the following code:

```
electrical a, b, a_b ; // Defines 3 nets, including a node named a_b  
V(a,b) <+ 2 * I(a,b) ; // An unnamed branch across a and b
```

Now, using the command `describe a_b` is ambiguous, because `a_b` could refer to the node `a_b` or to the unnamed branch. To resolve this ambiguity, Tcl assumes that `a_b` refers to the node and requires you to use `a_b_1` to refer to the unnamed branch. So you might have commands and responses like the following:

```
xcelium> describe a_b  
a_b.....analog net (electrical ) = 0  
xcelium> describe a_b_1  
a_b_1.....branch(a,b) = 0
```

It might sometimes be necessary to use additional generated names, such as `a_b_2`, if the names that would otherwise be used, such as `a_b_1`, are already in use. Generated names are used only for branches, never for nets.

To avoid the problem of ambiguous references, Cadence recommends that you declare and use named branches.

Related Topic

- [where](#)

Appendix C: Source Protection

Encryption allows you to protect your proprietary parameters, subcircuits, models, and netlists, and release your libraries to your customers without revealing sensitive information. Your customers can run simulations in the Spectre® circuit simulator with the encrypted netlists – the Spectre circuit simulator does not print any data inside the encrypted blocks. Messages about the encrypted portions of your circuit are suppressed.

Spectre AMS Designer simulator uses two different methods to protect (encrypt) source code, depending on what language you use. These two methods are similar but differ in terms of the commands you use and in the way they are implemented.

Language	Method for Protection
Verilog-A, Verilog-AMS, Verilog (digital), VHDL-AMS, and VHDL (digital) code	<code>xmprotect</code>
Spectre code	<code>spectre_encrypt</code>

You cannot probe any nets that go through an encrypted design unit.

You can use source protection with the AMS Designer simulator and analog solver configuration, Spectre solver using the simulation front end ([SFE](#)) parser.

Verilog-AMS Protection

It is not possible to partially protect a purely behavioral Verilog-AMS module or a Verilog-AMS module that contains both structural and behavioral information--if you protect anything in such a module, the entire module is protected. However, it remains possible to use Tcl or [VPI](#) to create a probe or to access values in the regions of the module that are not explicitly protected.

Protection Guidelines for Automatically Inserted Connect Modules

Automatically-inserted connect modules (AICMs) behave as follows:

- If the instance master for an AICM is protected, the instances of that master are also protected.

- If an AICM instance is connected to a protected net, the AICM instance is protected.
- If an AICM instance is within a protected instance, the AICM instance is protected.
- If an AICM instance is connected to an implicit net (a net that is undeclared), the AICM instance is protected if the net is protected. If the net is not protected, the AICM instance is not protected.
- All information related to a protected AICM instance is blocked.

Forced Use of CMI 3.0

You can use only a [CMI 3.0](#) device library or a device library compatible with CMI 3.0 in conjunction with source protection. Using an incompatible library in conjunction with source protection causes the simulator to stop immediately during initialization.

Related Topics

- [Using xmprotect for Source Protection](#)
- [Using spectre_encrypt for Source Protection](#)

Using xmprotect for Source Protection

When you use the `xmprotect` utility to prevent access to or modification of Verilog-AMS, Verilog (digital), VHDL-AMS, VHDL (digital), and Verilog-A source code, you can

- Protect selected design units or models
- Protect selected regions within design units or models
- Automatically protect all design units and models in a file

Source protection prevents access to protected regions. When you use source protection, software or commands that normally report information that depends on code do not return any information that might reveal the contents of the protected regions. In addition, AMS either suppresses warning and error messages from protected regions or issues generic messages that do not disclose protected information. You can use the protected code as usual in the simulation flow and it produces the same results as unprotected code.

To protect the source description of selected modules or regions:

1. Place protection pragmas in the source description to define the protected region.

The pragmas, which are in the form of comments, are:

- **`pragma protect`**: Indicates the start of a protection block. Used in conjunction with `pragma protect begin`.
- **`pragma protect begin`**: Indicates the start of the data to be encrypted.
- **`pragma protect end`**: Indicates the end of the data to be encrypted.

You use the protection pragmas to mark regions you want to protect in Verilog-AMS, Verilog (digital), VHDL-AMS, and VHDL (digital) code and Verilog-A code in your model files. See [Examples of Using the Protection Pragmas](#).

You can use the protection pragmas `protect begin` and `protect end` inside or outside of design units, provided that you pair each `protect begin` pragma with a `protect end` pragma in the same source file. If you insert a `protect begin` pragma without a corresponding `protect end` pragma, the software issues a warning and encrypts everything remaining in the file.

You can use multiple sets of the `protect begin` and `protect end` pragmas within design units. However, you cannot nest blocks of source code bounded by `protect begin` and `protect end` pragmas inside one another.

The following tasks do nothing when they are located inside an area that is protected: `$strobe`, `$fstrobe`, `$display`, `$fdisplay`, `$debug`, `$fdebug`, `$write`, `$fwrite`.

2. Run the `xmprotect` command on the input files containing the regions to be protected.

The protection pragmas mark the regions you want to protect; encryption actually occurs when you run the `xmprotect` command on the source description files.

Processing a source description with the `xmprotect` command generally protects only the regions marked with `protect begin` and `protect end` pragmas. The command creates a new source file that differs from the original file in the following ways:

- The pragmas `protect begin` and `protect end` become `protect begin_protected` and `protect end_protected`, respectively. The software adds other pragmas for the encryption.
- The regions you marked for protection in the original source description become unreadable.

By default, the new file has the same name as the original file, but with an appended p.

Ensure that the encrypted file is not changed after it is generated, perhaps by making the file read-only. Changing the encrypted code by hand corrupts the file, causing error messages such as the following:

```
Error while decrypting : Corrupted encrypted block, checksum did not match
```

If you get such an error, you can resolve the problem by recreating or reinstalling the protected code.

To use the protected modules, you run the compiler as usual. The compiler decrypts the encrypted files and compiles the design units in the file. You can then elaborate the design and simulate the snapshot. Downstream programs provide restricted visibility and access to the protected units.

Examples of Using the Protection Pragmas

These examples show how to use the `protect begin` and `protect end` pragmas in a source file.

The following example shows how to mark a region in the module `top_design` for protection:

```
module top_design (a, b, c)
    bottom inst ();
// pragma protect
// pragma protect begin
    initial
        $display ("Inside module top_design");
// pragma protect endendmodule
```

The protected version of the example takes the following form, allowing read access to the first two lines while encrypting the remainder of the module:

```
module top_design (a, b, c) // readable
bottom inst ();           // readable
//pragma protect begin_protected
//pragma protect key_keyowner=Cadence Design Systems.
//pragma protect key_keyname=CDS_KEY
//pragma protect key_method=RC5
//pragma protect key_block
hjQ2rsuJMpL9F3043Xx7zf656dz2xxBxdnHC0GvJFJG3Y5HL0dSoPcLMN5Zy6Iq+ySMMWcOGkowbtoHVjNn3Udc
ZFD6NF1WHJpb7KIc8Php8iTluEZmtwTgDSy64yqLLSCaqKffWXhnJ5n/936szbTSvc8vs2ILJYG4FnjIZeYARwK
jbofvTgA==
//pragma protect end_key_block
//pragma protect digest_block
```

```
uilUH9+52Dwx1U6ajpWVBgZque4=  
//pragma protect end_digest_block  
//pragma protect data_block  
jGZcQn3lBzXvF2kCXy+abmSjUdOfUzPOp7g7dfEzgN96O2ZRQP4aN7kqJOCA9shIjcvO6pnBhjaTNlxUJBSbBA=  
=  
//pragma protect end_data_block  
//pragma protect digest_blockt  
zEpxTPg7KWB9yMYylqfoVE3lVk=  
//pragma protect end_digest_block  
//pragma protect end_protected  
endmodule
```

The new, protected, source files do not overwrite the original, unprotected, source files. When you protect the original source file with `xmprotect`, you can specify an optional file extension you want the software to append to the name of the protected source file. If you do not specify an extension, the `xmprotect` command automatically appends a `p` to the source file name to create the protected file name.

For example, the following command protects the file `src.v`. By default, the software appends a `p` to the protected source file name: `src.vp`.

```
xmprotect src.v
```

The following command specifies an extension `myext` for the protected version of `design.v`:
`design.v.myext`.

```
xmprotect design.v -extension myext
```

If the name of the protected file conflicts with the name of an existing file, the `xmprotect` command does not create the protected file; instead, it issues a message that alerts you to the conflict.

The following example shows how to mark an entire module, including the module name, for protection:

```
// pragma protect  
// pragma protect begin  
    module bottom ();  
        initial  
            begin  
                $display ("Inside module bottom");  
            end  
        endmodule  
// pragma protect end
```

The following example illustrates how to protect a region in a VHDL description:

```
architecture behavior of myblock is
-- pragma protect
-- pragma protect begin
SIGNAL s: bit:= '0';

BEGIN
    digital: process (s)
    BEGIN
        s <= NOT s after 1 ms;
        REPORT "s=" & bit.image(s);
    END PROCESS digital;
END behavior;
-- pragma protect end
END myblock;
```

Related Topics

- [xmprotect Command Syntax and Options](#)
- [Using spectre_encrypt for Source Protection](#)

Protecting All Modules in a Source Description

To protect all modules in the specified source file automatically, you can use the `xmprotect -autoprotect` command (which you can use for Verilog-AMS, Verilog [digital], VHDL-AMS, and VHDL [digital] code and Verilog-A code but not for Spectre code). This option is particularly useful for protecting libraries that contain a large number of files with many modules.

You do not need to insert the `protect begin` and `protect end` protection pragmas in any source description that you plan to compile with `-autoprotect`. If these pragmas already exist in your source file, the `xmprotect -autoprotect` command ignores them.

Related Topic

- [Using xmprotect for Source Protection](#)

Using spectre_encrypt for Source Protection

`spectre_encrypt` is a standalone encryptor that you can use to encrypt Spectre model files. Encryption enables you to protect your proprietary parameters, subcircuits, models, netlists, and release your libraries to your customers without revealing sensitive information.

To encrypt part or all of a Spectre model file, do the following:

1. Open the file in a text editor.
2. Type `protect` above the data you want to protect.
3. Type `unprotect` after the last line of the data you want to protect. You must use the `protect` and `unprotect` keywords in pairs.

You use the `protect` and `unprotect` keywords to mark regions for protection in Spectre model files. For example, the following code leaves the name and I/O pins public.

```
subckt inv out in protect
parameters wi=1u le=3u
mp1 mid in vd vd pmos w=wi l=le
mn1 mid in 0 0 nmos w=wi l=le
r1 mid out resistor r=2k
model pmos bsim3v3 type=p tnom=27.0 tox=2.9e-09
model nmos bsim3v3 type=n tnom=27.0 tox=2.8e-09
unprotect
ends
```

For more information and examples, see "Encryption" in the *Spectre Circuit Simulator and Accelerated Parallel Simulator User Guide*.

4. Save the netlist.
5. Run `spectre_encrypt` on the input files containing the regions to be protected.

`xrun` can process design files that contain models encrypted using `spectre_encrypt` (when subcircuit boundaries are not encrypted). `xrun` cannot process subcircuits or **SPICE-in-the-middle** constructs encrypted using `spectre_encrypt` because of the encrypted boundaries.

The spectre_encrypt Command

Although the `protect` and `unprotect` keywords mark the regions of a Spectre model file that are to be protected, encryption actually occurs when you run the `spectre_encrypt` command on the source files.

Syntax

```
spectre_encrypt [-i input_file ][-o output_file ][-all]
```

where

<code>input_file</code>	The path and name of the file to be encrypted. If you do not specify the input file, the standard input is encrypted.
<code>output_file</code>	The path and name of a file to hold the encrypted model. The extension that you use for <code>output_file</code> must be the same extension used on <code>input_file</code> . If you do not specify the output file, the encrypted model is displayed as standard output in the terminal window.
<code>-all</code>	Encrypts the entire Spectre model file, ignoring any <code>protect</code> and <code>unprotect</code> keywords.

When you run `spectre_encrypt`, the `protect` and `unprotect` keywords are replaced with `pragma` statements in the output file. The `pragma` statements contain important information about encryption such as the method used, the key name, and the beginning and end of the encrypted block. The AMS Designer simulator uses this information while decrypting the netlist, so do not modify any `pragma` statements or the encrypted text between the `pragma` statements in the output file.

Related Topics

- [Using xmprotect for Source Protection](#)
- [Encryption](#)

Appendix D: Profiler for Performance Analysis

Profiler is an Xcelium tool that measures where CPU time is spent during simulation. The profiler generates a run-time profile file (`xmprof.out`) that contains simulation run-time information that is useful for finding performance bottlenecks and for tuning a design description for better simulation performance. This information can be used to modify your designs for better simulation.

You can invoke the profiler using the `-profile` option with `xrun`. When the simulator exits, the profiler writes the profile to the `xmprof.out` file in the current or run directories.

Each profile begins with a header that provides general information. The information contained in the header can reveal performance problems such as insufficient physical memory for the size of the simulation, or low CPU utilization due to a busy machine or waiting for I/O.

After the header, the profile is divided into three main sections: Mixed-signal simulation summary, Digital Simulation Profile Results, and Analog Simulation Profile Results. The tables in those sections share the following column headers:

<code>%hits</code>	The percentage of the total hits detected in the activity
<code>#hits</code>	The raw number of hits detected in the activity
<code>%cost</code>	The percentage of simulator time-steps required to service the analog operator or expression.
<code>domain</code>	The solver, either analog or digital
<code>#inst</code>	The number of instances
<code>name</code>	The name or description of the activity
<code>type</code>	One of the following types: <code>dVar</code> (a digital variable that causes a digital to analog event), <code>cross</code> (an <code>@cross</code> event), <code>filter</code> (a transition)
<code>instance</code>	The full hierarchical name of an instance and the line number where it is defined

The following are some examples of the simulation summary of mixed-signal, digital, and analog designs.

Mixed-Signal Simulation Summary

The mixed-signal simulation summary shows how simulation time divides between the analog solver and the digital solver. For example, the following report shows that 97% of the hits occurred in the analog solver.

```
-----  
Mixed-signal simulation summary (1766 hits total)  
-----
```

```
%hits #hits domain  
97.0 1713 Analog  
3.0 53 Digital  
-----
```

Digital Simulation Profile Results

The digital simulation profile results section summarizes information for the digital solver in three different ways: *Stream Counts*, *Most Active Modules (behavioral)*, and *Stream Type Summary Counts*.

Stream Counts

The *Stream Counts* section provides information about time spent in individual generated code streams. These code streams correspond to specific HDL source constructs. They include

- `always` blocks
- `initial` blocks
- Continuous assignments
- Tasks and functions
- Non-blocking assignments
- Quasi-continuous assignments
- Parallel block sub-processes (statements inside `fork/join`)
- Logic primitives for gates and UDPs (because logic primitives share the same code between different instances, the profiler cannot indicate which gate instances are taking up the most time)

The beginning of the *Stream Counts* section is the first place to look for inefficiency. If a few streams

are taking up most of the simulation time, simulation cannot be sped up significantly without reducing the time in those streams.

The following is an example of a *Stream Counts* section.

```
-----
Stream Counts (53 hits total)
-----

%hits #hits #inst name
28.3      15 [   ] Library function minnow (time callbacks)
24.5      13 [   ] tcl_functions
15.1       8 [   ] Method SSS_KM_CL2TA (method)
15.1       8 [   ] outside engine
 5.7       3 [   ] Method SSS_KM_FINDRFT (method)
 3.8       2 [  1] Always stmt (file:
/net/mach111/cds/ams/ldv51/tools/affirma_ams/etc/connect_lib/elect2logic.v, line: 69 in
connectLib.elect2logic      [module])
 3.8       2 [   ] Library function rtl_analog_demand
 1.9       1 [   ] Method SSS_MT_SA_PROCESS (method)
 1.9       1 [   ] ssslib snare support
```

The example of the *Stream Counts* section shown above contains a category called `outside engine`. This category, and another called `engine support`, are catch-all categories for activities that cannot be otherwise categorized.

Most Active Modules

The *Most Active Modules* section of the profile summarizes the stream counts by module. For each module listed, the profile lists the sum of the counts in all of the streams.

The following example shows a *Most Active Modules* section.

```
-----
Most Active Modules (behavioral)
-----

%hits #hits #inst name
 3.8      2 [  1] connectLib.elect2logic:module (file:
/net/mach111/cds/ams/ldv51/tools/affirma_ams/etc/connect_lib/elect2logic.v line: 60)
```

Stream Type Summary Counts

The *Stream Type Summary Counts* section summarizes the stream counts by the type of stream or other activity. For example, there might be a total for logic primitives, timing checks, `always` or `initial` statements, non-blocking assignments, continuous assignments, and so on.

The summary makes it easier to identify widespread inefficiencies in the simulation. For example, large amounts of time spent on probing, file I/O, and [PLI](#) show up most clearly in this section.

The following example shows a *Stream Type Summary Counts* section:

```
-----
Stream Type Summary Counts (53 hits total)
-----

%hits #hits #inst name
32.1      17 [   ] System tasks/functions or library functions
24.5      13 [   ] VCD/SHM variable dumping
22.6      12 [   ] Standard methods (mostly fanout propagation)
15.1       8 [   ] Outside engine
 3.8       2 [  3] Always statements
 1.9       1 [   ] Support for VPI callbacks (or UI)
```

Analog Simulation Profile Results

The *Events and Operators* section of the *Analog Simulation Profile Results* reports performance measurements for analog constructs, including, for Verilog®-AMS

- `transition`
- `@cross`
- Accessing digital variables and nets from an analog context

and including, for VHDL-AMS

- `S'ramp`
- `Q'above`
- Accessing digital variables and nets from an analog context, with or without `break` statements.

The following example shows an *Events and Operators* section.

```
----- Events and Operators -----
```

```
%cost #hits type instance
4.4    135 dVar      top.dac:dacOut (file: AMS_lib/daconv.vhd line: 22)
4.4    135 dVar      top.dac:dacOut'DELAYED (file: AMS_lib/daconv.vhd line: 37)
4.4    135 filter    top.dac (file: ../../../../../../../AMS_lib/daconv.vhd line: 37)
0.5     15 filter    top.sh (file: AMS_lib/samplehold.vams line: 38)
```

Related Topic

- [xrun Command-Line Options for Mixed-Signal Designs](#)

Appendix E: Migration of prop.cfg file to an amsd Block

If you are migrating from using a `prop.cfg` file to using an `amsd` block, you must map `prop.cfg` statements to the corresponding `amsd` block statements. The following are the specifications you must modify:

- [string prop sourcefile Translation](#)
- [cell, inst, and path Translations](#)
- [Stub Instance Translations](#)
- [default Translations](#)
- [string prop hdl_cell Translation](#)
- [string prop sim_stub Translation](#)
- [string prop sourcefile_opts Translations](#)

string prop sourcefile Translation

Instead of declaring the `sourcefile` string property in a `prop.cfg` file, use an `include` statement in the file containing the `amsd` block:

prop.cfg Syntax	File Containing amsd Block
<pre>string prop sourcefile " file "</pre>	<pre>include " file " amsd{...}</pre>

cell, inst, and path Translations

cell

To declare a cell in the `amsd` block, use the `portmap` and `config` statements as follows:

prop.cfg Syntax	amsd Block Syntax
cell myCell	portmap subckt=myCell config cell=myCell use=spice

inst

prop.cfg Syntax	amsd Block Syntax
inst I0 {...}	The <code>amsd</code> block does not support the <code>inst</code> keyword (<code>prop.cfg</code>) directly. Instead, you can use occurrence-based instance paths. For example, instead of <pre>inst I0 {...}</pre> consider <pre>path top.I0 {...}</pre> for which the <code>amsd</code> block equivalent is <pre>portmap subckt=bar_subckt config inst=top.I0 use=spice</pre>

path

The `path` keyword (`prop.cfg`) translates to the `inst` keyword on a `config` statement in the `amsd` block, using explicit path names, as follows:

prop.cfg Syntax	amsd Block Syntax

path top.I0	portmap subckt=bar_subckt config inst=top.I0 use=spice
path (top).s2.t1	config inst=top.s2.t1 The SFE parser does not support the use of parentheses in statements in the amsd block , so (top).s2.t1 becomes top.s2.t1

Stub Instance Translations

prop.cfg Syntax	amsd Block Syntax
<pre> path (top).xana_top2{ string prop sim_stub = "-src analog_top.cir -cell analog_top"; } path (top).xana_top3 { string prop sourcefile="analog_top.cir"; string prop sourcefile_opts="- auto_bus -subckt analog_top"; } path (top).xana_top4 { string prop sim_stub="worklib.analog_top:module"; } </pre>	<pre> include "analog_top.cir" amsd{ // example of spice stub portmap stub=analog_top match=spice // Note new card:stub=cellname, match=spice config inst=top.xana_top2 use=stub // spice subckt portmap subckt=analog_top autobus=yes config inst=top.xana_top3 use=spice // verilog stub portmap stub=analog_top match=verilog // stub=cell & match=verilog config inst=top.xana_top4 use=stub } </pre>

default Translations

You can specify default cell and bus settings using the `config` and `portmap` statements:

prop.cfg Syntax	amsd Block Syntax
<pre>default hdl_cell="nand2"</pre>	<pre>config cell=nand2 use=hdl</pre>
<pre>default sourcefile_opts_common= "-auto_bus -bus_delim []"</pre>	<pre>portmap autobus=yes busdelim="[]"</pre>

string prop hdl_cell Translation

The `hdl_cell` property (`prop.cfg`) translates to the `use=hdl` specifier on a `config` statement in the `amsd` block as follows:

prop.cfg Syntax	amsd Block Syntax
<pre>string prop hdl_cell="tb"</pre>	<pre>config cell=tb use=hdl</pre>
<pre>string prop hdl_cell="cell11 cell12"</pre>	<pre>config cell="cell11 cell12" use=hdl</pre>

string prop sim_stub Translation

The `sim_stub` property (`prop.cfg`) translates to the `use=stub` specifier on a `config` statement in the `amsd` block as follows:

prop.cfg Syntax	amsd Block Syntax
<pre>inst xana_top2 sim_stub="- cell analog_top -src analog_top.scs"</pre>	<pre>portmap stub=analog_top match=spice config inst=xana_top2 use=stub</pre>

<pre>path (top).xana_top2 sim_stub= "worklib.analog_top:module"</pre>	<pre>portmap stub=worklib.analog_top:module match=verilog config inst=top.xana_top4.I0 use=stub</pre>

See also [Stub Instance Translations](#).

string prop sourcefile_opts Translations

The `sourcefile_opts` properties (`prop.cfg`) translate to `amsd` block syntax as follows:

prop.cfg Syntax	amsd Block Syntax
<code>sourcefile_opts="-auto_bus"</code>	<code>portmap autobus=yes, or no autobus specification (autobus=yes is the default)</code>
<code>sourcefile_opts="-bus_delim <>"</code>	<code>portmap busdelim="<>"</code>
<code>sourcefile_opts="-cell_case keep"</code>	<code>portmap ... cellcase=keep</code>
<code>sourcefile_opts="-case_map upper"</code>	<code>portmap casemap=upper</code>
<code>sourcefile_opts="-exclude_bus=itunea"</code>	<code>portmap excludebus=itunea</code>
<code>sourcefile_opts="-in_port in1*in1"</code>	<code>portmap input="in1*in1"</code>
<code>sourcefile_opts="-input mica" or sourcefile_opts="-input ti"</code>	No special entry in <code>amsd</code> block. Instead, use <code>-amsi mica:file_mica.cir</code> or <code>-amsi ti:file_ti.cir</code> on the <code>xmelab</code> command line.

sourcefile_opts="- input spice"	use=spice
sourcefile_opts="- no_bus"	portmap autobus=no
sourcefile_opts="- out_port itunea	portmap output=itunea
sourcefile_opts="- portmap_file analog_top.pb"	portmap file="analog_top.pb "
sourcefile_opts="- reverse_bus itunea"	portmap reversebus=itunea
sourcefile_opts="- subckt buf_array_sp"	portmap subckt=buf_array_sp
sourcefile_opts="- veri_file analog_top.v"	portmap reffile="analog_top.v" reformat=verilog
Escaped names: path (\l_wrap).I1	config inst="top.I4.1_I1" Use the actual unescaped name.

Glossary

A

ADE

Abbreviation for Analog Design Environment. The Cadence® Virtuoso® Analog Design Environment is the analog design and simulation environment for the Virtuoso custom design platform. It has become an industry's standard environment for simulating and analyzing full-custom, analog, and RF IC designs, and it is the task-based tool within the Virtuoso Specification-driven Environment.

AICM

Abbreviation for Automatically-inserted Connect Module.

access function

The method by which flows and potentials are accessed on nets, ports, and branches.

analog procedural block

A procedural sequence of statements that defines the behavioral description of a continuous time simulation.

B

BSIM

Abbreviation for Berkeley Short-channel IGFET Model. BSIM is a physics-based, accurate, scalable, robust and predictive MOSFET SPICE model for circuit simulation and CMOS technology development. It is developed by the BSIM Research Group in the Department of Electrical Engineering and Computer Sciences (EECS) at the University of California, Berkeley.

branch

A path between two nodes. Each branch has two associated quantities, a potential and a flow, with a reference direction for each.

C

CM

Abbreviation for connect module. A module inserted automatically or manually by using the `connect` statement, which contains the code required to translate and propagate signals between nets that have different discipline domains and that are connected through a port. Connect modules are also known as interface elements.

CMI

Abbreviation for Compile Module Interface. A Cadence API used to include model primitives in the C and C++ languages. CMI is shipped with the SPECTRE release. The CMI models can be used in Spectre, APS, UltraSim, XPS, and AMS-D simulators.

circuit topology

The interconnection of all the circuit elements with given parameters and port bindings.

collapsible and non-collapsible port connection

A port connection is collapsible if the upper and lower connections are nets. For connections of selects of packed or unpacked net arrays, the selects must have constant indices to be collapsible. It is important to note that, amongst other things, the presence of variables or constant expressions on either side of the connections makes the port connections non-collapsible.

D

DSPF

Abbreviation for Detailed Standard Parasitic Format. A file format to represent parasitic data.

DUT

Abbreviation for Design Under Test. Typically DUT refers to the portion of the simulation that is synthesized. The other portion of the simulation could be referred to as the test bench.

digital island

The set of drivers and receivers interconnected by a purely digital net.

discipline resolution

The process of assigning a domain and discipline to nets whose domain and discipline are otherwise unknown (or whose discipline is wire).

driver

A primitive device or behavioral construct that affects the digital value of a signal.

driver-receiver segregation

The conceptual severing of connections between drivers and receivers that occurs in mixed nets. When driver-receiver segregation occurs, digital signals propagate only through connect modules inserted between the drivers and receivers.

E

E2R

Abbreviation for Electrical-to-Real. An automatically-inserted connect module that connects a Verilog-AMS electrical object and a SystemVerilog Real object (wreal), and converts a signal with electrical discipline to logic real value.

G

GUI

Abbreviation for Graphical User Interface.

H

HDL

Abbreviation for Hardware Description Language. HDL is any language from a class of computer, specification, or modeling languages used for formal description and design of electronic circuits and digital logic. It can describe a circuit's operation, design and organization, and tests to verify its operation through simulation.

HED

Abbreviation for Hierarchy Editor. The Cadence hierarchy editor (HED) is used in the Cadence® Virtuoso flow to define the design partitioning during analog and mixed-signal simulation. The defined design partitioning is stored in a configuration file.

I

IE

Abbreviation for Interface Element. Also known as connect modules, interface elements work as analog-to-digital (A2D) and digital-to-analog (D2A) converters during mixed-signal simulation. IEs operate the electrical-to-logic and logic-to-electrical conversions. The Spectre® AMS Designer Simulator allows automatic insertion of IEs during elaboration.

M

mixed bus

A bus comprising at least one net from the analog domain and at least one net from the digital domain.

MOSFET

Abbreviation for Metal-Oxide-Semiconductor Field-Effect Transistor. MOSFET is a transistor used for amplifying or switching electronic signals. In MOSFET, a voltage on the oxide-insulated gate electrode can induce a conducting channel between the two other contacts called source and drain. The channel can be of n-type or p-type and is accordingly called an nMOSFET or a pMOSFET (also commonly known as nMOS and pMOS).

MSDC

Abbreviation for Mixed-Signal DC.

mtline

Abbreviation for multi-conduction transmission line. Characterized by constant RLGC matrices or frequency-dependent RLGC data, an mtline can contain as many conductors as described in the input. However, there must be at least two conductors, with one conductor used as a reference to define terminal voltages. The reference conductor can be ground. The order of the conductors is the same as the order of the data in the input. The mtline model is included in Spectre, APS, UltraSim, XPS and AMS-D simulators.

MTS

Abbreviation for Multi-technology Simulation. Technology to enable the simulation of a system that consists of IC blocks to be manufactured with different processes.

N

NCF

Abbreviation for Netlist Compiled Function. The Spectre circuit simulator allows a netlist expression to call functions that are loaded from a Dynamic Link Library (DLL). By creating functions in C or C++, for example, it takes advantage of the features of these languages and overcomes the restrictions of the netlist user-defined function.

xmelab

Native code elaborator. The elaborator gathers various portions of a design and creates a snapshot that can be simulated (with xmsim). See [Elaborator Options](#).

xmsim

Native code simulator. See [Simulator Options](#).

xmvhdl

Native code VHDL compiler. See [Introduction to the Xcelium Executables](#).

xmvlog

Native code Verilog compiler. See [Introduction to the Xcelium Executables](#).

O

OMI

Abbreviation for Open Model Interface. An IEEE 1499 standard, which ensures open exchange of HDL-based IP. It is a language-neutral interface between models and simulation tools.

ordinary module

Any module other than a connect module.

OOMR

Abbreviation for Out-of-module reference. A direct reference from one Verilog module to another module that does not pass through any ports.

P

.pak

The NC Simulator stores output data from xrun, xmvlog, xmvhdl, and xmelab in one or more binary object files with the extension `.pak`, in the library in which the files are compiled.

PLI

Abbreviation for Programming Language Interface. A procedural interface that allows C/C++ functions to access the internal data structures of a SystemVerilog simulation. PLI includes SystemVerilog/Verification Procedural Interface (VPI) and VHDL Procedural Interface (VHPI).

PPE

Abbreviation for Post-processing Environment. Refers to the environment where simulation has finished and the results are being analyzed typically with a waveform viewer, as opposed to "live-simulation," where simulation is still running.

PSF

Abbreviation for Parameter Storage Format. When the AMS Designer simulator runs standalone, it writes the results of the AC analysis to a PSF file. By default, the software stores the PSF file in a

directory called ascf.raw (where ascf is the name of the analog simulation control file).

PSL

Abbreviation for Property Specification Language. An IEEE 1850 standard for Property Specification Language.

PSO

Abbreviation for Power-shutoff. PSO, also called power gating, is one of the most effective power management techniques for reducing power. In PSO, selected functional blocks of the chip are individually powered down when they are not in use, to save leakage and dynamic power.

R

R2E

Abbreviation for Wreal-to-Electrical. A connect module, also known as an interface element, which converts digital logic values (0, 1, x, and z states) to real numbers.

receiver

A primitive device or behavioral construct that samples the digital value of a signal.

RNM

Abbreviation for Real Number Modeling or Real Number Models. Technique used to speed up simulation. As compared to SPICE and Fast SPICE simulators, RNM is more effective as it uses digital solvers (instead of analog solvers), and real ports and real variables in the behavioral code. During RNM simulation, the analog continuous blocks are represented with discrete real values computed in a digital solver. RNM simulations provide several magnitudes of simulation speed as compared to analog simulations. Though traditional analog verification flows provide accurate results, they are slower than RNM simulations. This is because analog solvers compute non-linear differential equations with matrixes, and such computations consume time and cost huge amount of CPU computations. Thus, unlike SPICE simulations, RNM displays high simulation speed gain, facilitates fast computation, and does not display convergence issues. RNM performance enables co-simulation of hardware and software in mixed-signal SoC verification.

RTSF

Abbreviation for Rain Tree Storage Format. Cadence proprietary format created by the Spectre, APS, UltraSim, and AMS-D simulators. RTSF is a PSF XL extension that provides improved viewing performance in the Virtuoso Visualization and Analysis XL tool. RTSF facilitates ultra fast viewing of data that contain a large number of time points.

S

SFE

Abbreviation for Simulation Front End.

SHM

Abbreviation for Simulation History Manager. A high-performance Cadence proprietary database for waveforms and related data used by NC-Sim, SimVision, ViVA, and other tools. It consists of a directory, typically with a ".shm" suffix, containing one or more SST2 database files.

singular interconnect

A singular interconnect is an interconnect which has no bounds declared. It represents a single atomic connection.

SPEF

Abbreviation for Standard Parasitic Exchange Format. An IEEE standard for representing parasitic data related to wires in a chip, in the ASCII format.

SST2

A Cadence proprietary database format used to store waveforms and related data in an SHM database directory. SST2 database files use the suffixes ".dsn," ".trn," and ".stc."

SV

Abbreviation for SystemVerilog. An IEEE P1800 standard for unified hardware design, specification, and verification language.

SVA

Abbreviation for SystemVerilog Assertion.

signal

A hierarchical collection of nets which, because of port connections, are contiguous.

T

topology

See [circuit topology](#) .

topology changes

Removal or introduction of nodes or using different devices in a design as a result of replacing subcircuit definitions or Verilog-A modules with respect to the original Spectre/SPICE files

submitted to `xmsim` for simulation.

U

UDP

Abbreviation for User-defined Primitives. Allows designers to create objects during the modeling task. Multiple languages, such as Verilog and VHDL, support UDPs.

V

VCD

Abbreviation for Value Change Dump. A waveform format that SimVision can use; its format is ASCII-based and non-proprietary, allowing it to be used by many tools from different companies.

VPI

Abbreviation for SystemVerilog/Verification Procedural Interface. VPI provides a library of C-language functions and a mechanism for associating foreign language functions with SystemVerilog user-defined system task and system function names. SystemVerilog also has a DPI protocol for interacting with C code.

General Terms Related to Simulation

ABV

Abbreviation for Assertion Based Verification. A methodology based on the assertion properties defined by the designer, to verify circuit operation. The assertion properties check or capture the design intent. Usually when the property check finds a failure, it reports an error. ABV allows improvisation of the design quality and verification productivity. Some languages according to the IEEE standard such as PSL and SVA are focused on ABV.

AXUM

Abbreviation for AMS Designer Xcelium Use Model. AXUM is a use model of AMS Designer dedicated for digital centric users, based on command-line simulation and SimVision debugging environment.

AVUM

Abbreviation for AMS Designer Virtuoso Use Model. A Cadence® Virtuoso® Analog Design Environment (ADE) based AMS Designer use model dedicated for analog- and mixed-centric users. The main difference between the AVUM and the AXUM flows is that AVUM is schematic based. It uses HED for design (analog/digital) partitioning. ADE provides netlisting, and the powerful and advanced simulation and post-processing cockpit.

AST

A VHDL syntax tree; an output data object type. NC and AMS-D simulators store such output data object type in the `.pak` file.

COD

Refers to code; is an output data object type. NC and AMS-D simulators store such output data object type in the `.pak` file.

DFII

Abbreviation for Design Framework II. Refers to the former Cadence Design Framework II, which has now become Virtuoso Design Environment in IC 6.1.

MDV

Abbreviation for Metric Driven Verification. Key methodology to silicon realization. MDV helps the design creators and integrators close the productivity gap (through improved approaches to design, verification, and implementation) and the profitability gap (by providing new capabilities for system optimization and IP creation, selection, and integration). MDV broadens the scope of the term "metrics" by including checks, assertions, and software- and time-based data points.

OA

Abbreviation for Open Access. A database format. OpenAccess data is supported by many industry-leading physical design tools, including the Cadence Virtuoso Custom Design Platform, as it helps eliminate tedious translation steps to save time and minimize misstep.

SAM

Abbreviation for Simulation Analog Master. An output data object type. NC and AMS-D simulators store such output data object type in the `.pak` file.

SIG

Overlay tables; an output data object type. NC and AMS-D simulators store such output data object type in the `.pak` file.

SSS

Abbreviation for Simulation Snapshot. An output data object type. NC and AMS-D simulators store such output data object type in the `.pak` file.

VHPI

Abbreviation for VHDL Procedural Interface. An interface that allows C-language programs access to VHDL design information and provides foreign language functions along with the ability to interact with a VHDL simulator through VHDL code.

VST

Abbreviation for Verilog Syntax Tree. An output data object type. NC and AMS-D simulators store such output data object type in the `.pak` file.