cādence®

Virtuoso® AMS Designer Simulator User Guide

Product Version 13.2
January 2014

Contents

About This Manual

You can find information about Related Documents and Typographic and Syntax Conventions in this preface.

Related Documents

For more information about the AMS Designer simulator and related products, consult the sources listed below.

- *Cadence Hierarchy Editor User Guide*
- *Cadence Library Manager User Guide*
- *Cadence Verilog-A Language Reference*
- *Cadence Verilog-AMS Language Reference*
- *Component Description Format User Guide*
- *IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS Changes), IEEE Std 1076.1*. Available from IEEE.
- *Instance-Based View Switching Application Note*
- *Overview of Running the Incisive Enterprise Simulator*
- *Introduction to SimVision*
- *Verilog-A Debugging Tool User Guide*
- *Verilog-AMS Language Reference Manual*. Available from Open Verilog International.
- *Verilog-XL Reference*
- *Virtuoso AMS Environment User Guide*
- *Virtuoso ADE L User Guide*
- *Virtuoso Mixed-Signal Circuit Design Environment User Guide*
- *Virtuoso Schematic Editor L User Guide*

- *Virtuoso Spectre Circuit Simulator Reference*
- *Virtuoso Spectre Circuit Simulator User Guide*
- *Virtuoso UltraSim Simulator User Guide*

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 lower-case 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 ;
  ```

1 Product and Licensing Information

To run the AMS Designer simulator, you must have access to a corresponding license or combination of licenses. The order in which these licenses are used is determined either by default or by using the `-uselicense` option.

The `-uselicense` option lets you specify a custom license checkout order for simulation. AMS Designer checks for a license in the specified order.

```
-uselicense mnemonicList[:DEFAULT]
```

See the following topics for more information:

- Elaboration License Checkout Order on page 25
- Default License Checkout Order on page 26
- Feature-to-License Checklist on page 27
- Valid Mnemonics and License Strings on page 28
- AMS Designer Licensing Products on page 31
- Valid Mnemonics and License Strings Related to AMS Designer Verification Option on page 34

Elaboration License Checkout Order

Block-based discipline resolution, discrete discipline compatibility checking, and programmable DMS coercion require an elaboration license, for which, the default checkout order is:

```
DMSO:AMSDL:MMSIM2
```

Note: The use of an `ie` card on a pure DMS simulation can require an elaboration license, depending on the use case.

Default License Checkout Order

The default license checkout order for the AMS Designer simulator is as follows:

`AMSDL:MMSIM2:IESXLMMSIM`

Note: AMSDL is the first item in the license order; so `ncsim` would try to check this out first.

Exceptions to the above-mentioned default licensing order include the following:

- If you are using AMS Designer simulator with Incisive advanced digital verification capabilities such as System C and Cover Groups, the required license is IESXLMMSIM.
- Analog solvers check out the required license or MMSIM tokens for the solver you are attempting to run.
- If you are using SV-Real randomization, the licenses are checked out in the following order:

  1. IES-XL + Digital_Mixed_Signal_Option
  2. AMS Designer Link + 1 MMSIM token
  3. Three MMSIM tokens

Feature-to-License Checklist

The following table illustrates the mapping of feature groups to licenses.

Feature	License Strings
Basic `wreal` features (see Basic wreal Features of the AMS Designer Simulator on page 232)	Incisive_Enterprise_Simulator (IES-XL) Digital_Mixed_Signal_Option (DMS) AMS_Designer_Verification **Note:** These features, when used in a pure digital simulation, will require an IES-XL license, or three MMSIM tokens, or an equivalent combination. The license order used should be: `IES:WREALMMSIM1:WREALMMSIM3` If an analog solver is used, the licensing requirement will be determined based on the other features.
Advanced `wreal` features (see Advanced wreal Features of the AMS Designer Simulator on page 233)	Incisive_Enterprise_Simulator (IES-XL) Digital_Mixed_Signal_Option (DMS) AMS_Designer_Verification **Note:** These features, when used in a pure digital simulation, will require IES-XL and DMS licenses, or three MMSIM tokens, or an equivalent combination. The license order used should be: `IESXLDMS:WREALMMSIM1:WREALMMSIM3` If an analog solver is used, the licensing requirement will be determined based on the other features.
Advanced AMS Designer features (see AMS Designer Verification Option on page 460)	AMS_Designer_Verification

Feature	License Strings
Advanced design-dependent features	Incisive_Enterprise_Simulator (IES-XL)
	Digital_Mixed_Signal_Option (DMS)
	AMS_Designer_Verification
	Note: If only a digital solver is being used by the design, you will need IES-XL and DMS licenses. If an analog solver is used, an `AMS_Designer_Verification` license will be required.

Valid Mnemonics and License Strings

The following table lists the set of valid mnemonics and corresponding license strings for the Virtuoso® AMS Designer Simulator. **(M)** indicates a master feature string. Where the simulator requires more than one license token, the required number appears as **[tokens:** *`numTokens`***]** after the license string.

License Mnemonic	License Strings
AMSDL	AMS_Designer_Link
IESXLDMS	Incisive_Enterprise_Simulator Digital_Mixed_Signal_Option
	Note: The Digital_Mixed_Signal_Option (DMS) license is required for designs that use only digital solvers. If the design uses an analog solver, DMS cannot be used.
MMSIM2	Virtuoso_Multi_mode_Simulation [tokens: 2]
MMSIM4	Virtuoso_Multi_mode_Simulation [tokens: 4]
IESXLMMSIM	Incisive_Enterprise_Simulator Virtuoso_Multi_mode_Simulation
IESXLMMSIM3	Incisive_Enterprise_Simulator Virtuoso_Multi_mode_Simulation [tokens: 3]
SPECTRE	*Use the license/token checkout rules for Spectre*
USIM	*Use the license/token checkout rules for UltraSim*

License Mnemonic	License Strings
WREALMMSIM1	AMS_Designer_Link Virtuoso_Multi_mode_Simulation
WREALMMSIM3	Virtuoso_Multi_mode_Simulation [tokens: 3]
MMSIMLK	Virtuoso_Spectre_GXL_MMSIM_Lk

Depending on your requirement, you can choose any combination of the above-mentioned mnemonics. The software checks out the set of licenses that satisfies the license requirement and continues.

The `Virtuoso_Spectre_GXL_MMSIM_Lk` license sring is checked out for every MMSIM token utilized by the AMS Designer simulator, or the analog solver that is invoked. This license string is optional. If it is not available, the simulation run is not affected.

You must be using AMS Designer with Flexible Analog Simulation licensing (which corresponds to the AMSDL license mnemonic and the AMS_Designer_Link license string) in order to access Spectre, APS, or UltraSim product features.

In the following example, the software first attempts to check out an AMS_Designer_Link license (which corresponds to the AMSDL license mnemonic), then licenses associated with the MMSIM2 mnemonic, in that order. If the software finds that these mnemonics do not provide sufficient rights or if it cannot check out the associated licenses successfully, it switches to the default license checkout order.

```
ncsim -uselicense AMSDL:MMSIM2:DEFAULT …
```

In the following example, the software first attempts to check out two Virtuoso_Multi_mode_Simulation license tokens (which corresponds to the MMSIM2 license mnemonic) for the AMS part, then further checks out Virtuoso_Multi_mode_Simulation license tokens according to the license/token checkout rules for the analog solver.

```
ncsim -uselicense MMSIM2
```

In the following example, the software first attempts to check out two Virtuoso_Multi_mode_Simulation license tokens (which corresponds to the MMSIM2 license mnemonic) for the AMS part, then further checks out Virtuoso_Multi_mode_Simulation license tokens according to the license/token checkout rules for the analog solver. When the MMSIM2 mnemonic provides sufficient licenses for the run, the software does not go on to use the SPECTRE or USIM mnemonic.

```
ncsim -uselicense MMSIM2:SPECTRE
ncsim -uselicense MMSIM2:USIM
```

In the following example, the software skips over the SPECTRE mnemonic during a first pass check for AMS licensing, and checks out two Virtuoso_Multi_mode_Simulation license tokens (which corresponds to the MMSIM2 license mnemonic) for the AMS part. Then, for the analog solver license checkout, the software uses the SPECTRE mnemonic to check out licenses/tokens for Spectre following default license/token checkout rules for Spectre. If Spectre licenses are insufficient, the software checks out the appropriate number of Virtuoso_Multi_mode_Simulation license tokens.

```
ncsim -uselicense SPECTRE:MMSIM2
```

In the following example, the software tries to check out licenses associated with the AMSDL mnemonic and then with the MMSIM2 mnemonic.

```
ncsim -uselicense AMSDL:MMSIM2 lib.cell:view
```

AMS Designer Licensing Products

The following table provides the descriptions of the two key AMS Designer licensing products.

Product # / Name	License Mnemonic	Description
70020 Virtuoso AMS Designer	AMSDL	Virtuoso AMS Designer is the platform on which the Cadence Mixed-Signal Simulation solution is built. AMS Designer simulator enables mixed-signal capabilities with several analog simulation options, enabling mixed-signal design and verification. **Note:** AMSDL supports `ncelab` options such as `setdiscipline` and `chkdigdisp`. Spectre and Spectre APS are licensed separately. Some of the features enabled by the AMS Designer's analog and digital simulation capabilities are: ■ Automatic insertion of interface elements ■ Verilog-AMS and VHDL-AMS language support ■ Basic and advanced wreal ■ Bi-directional ports ■ SystemVerilog design

Product # / Name	License Mnemonic	Description
70030 Virtuoso AMS Designer Verification Option	AMSDVER	Virtuoso AMS Designer Verification Option provides a complete solution for mixed-signal SoC verification by enhancing the performance and capacity of existing AMS block-level technology, supporting cross-domain connectivity between test benches and IP from multiple vendors, and extending mature digital verification methodologies such as low-power to the analog domain. Some of the major features offered by this license are: ■ CPF Low-power ❑ Power shut-off support ❑ Power modes and nominal condition support (CPF1.1 compliance) ❑ Analog value fetch function `cds_get_analog_value` or `cgav` ❑ Analog fetch helper functions for `cgav` ❑ CPF control analog ❑ Power aware L2E/E2L IE ❑ Isolation rule on mixed signal boundary ❑ Support for X-state changes in wreal boundary port in macro model ❑ Support for wreal expressions in CPF

Product # / Name	License Mnemonic	Description
		■ VHDL-SPICE ❑ SPICE on leaf ❑ SPICE in middle ❑ CE (Conversion Element) Optimization ■ SystemVerilog ❑ Real number modeling ❑ Testbench constructs ■ Mixed-signal Stitching ■ AMS Assertions ❑ PSL on electrical nets ❑ Analog value fetch function `cds_get_analog_value` or `cgav` ❑ Analog fetch helper functions for `cgav` ■ AMS Linter

Valid Mnemonics and License Strings Related to AMS Designer Verification Option

The following table lists the set of valid mnemonics and corresponding license strings related to AMS Designer Verification option, which provides the advanced AMS Designer features. Where the simulator requires more than one license token, the required number appears as **[tokens:** *numTokens***]** after the license string.

License Mnemonic	License Strings
AMSDVER	AMS_Designer_Link AMS_Designer_Verification
AMSDVERMMS2	AMS_Designer_Verification Virtuoso_Multi_mode_Simulation [tokens: 2] Virtuoso_Spectre_GXL_MMSIM_Lk [tokens: 2
IESXLMMS1AMSDVER	Incisive_Enterprise_Simulator Virtuoso_Multi_mode_Simulation AMS_Designer_Verification Virtuoso_Spectre_GXL_MMSIM_Lk

If the AMS Designer simulator is running using the AMS Designer Verification option for advanced analog and mixed-signal features such as: Mixed Signal Stitching, Analog Assertions, VHDL-SPICE, and CPF, the default mnemonics will be changed to:

`AMSDVER:AMSDVERMMS2:MMSIM4`

If the AMS Designer simulator is running using the AMS Designer Verification option for advanced digital verification capabilities such as: SystemVerilog real number modeling, and SystemVerilog Testbench, the default mnemonics will be changed to:

`AMSDVER:AMSDVERMMS2:MMSIM4:IESXLMMS1AMSDVER`

2

Getting Started with the AMS Simulator

The Virtuoso® AMS Designer simulator is a mixed-signal simulator that supports the Verilog®-AMS language standard and simulation speed with more than 50K elements in SPICE blocks.

Tip

Virtuoso AMS Designer Simulator Tutorials provides information about how to perform certain design tasks using the Virtuoso® AMS Designer simulator.

See the following topics for more information:

- Language Support on page 36
- Running the Virtuoso AMS Designer Simulator on page 37
  - Running the Simulator in a Single Step on page 38
  - Running the Simulator in Three Steps on page 39
- Checking Analog Solver-Related Information on page 47
- Using Advanced Analog Solver Features on page 47

See also

- More about UltraSim Features on page 49
- *Virtuoso Spectre Circuit Simulator User Guide*
- *Virtuoso UltraSim Simulator User Guide*

Language Support

Except as noted, the Virtuoso® 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

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

For information on language features not supported by the Virtuoso AMS Designer simulator, see “Unsupported Elements of Verilog-AMS” in the *Cadence Verilog-AMS Language Reference*.

Running the Virtuoso AMS Designer Simulator

There are two ways to run the Virtuoso AMS Designer simulator:

- Single-step (see "Running the Simulator in a Single Step" on page 38)

  In this approach, you issue one command, the `irun` command. This command automatically runs the three steps in turn: the `ncvlog` compiler, the `ncelab` elaborator, and the `ncsim` simulator. Cadence recommends this easier-to-use use model.

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

- Three-step (see "Running the Simulator in Three Steps" on page 39)

  In this approach, you run `ncvlog`, `ncelab`, and `ncsim` separately.

Important

For the three-step flow, you need library path environment variable as follows:

For the 64-Bit Version

For Red Hat Linux, set your 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 your library path environment variable as follows:

```
setenv LD_LIBRARY_PATH install_dir/tools/lib/64bit:install_dir/tools/lib/
64bit/SuSE:install_dir/tools/systemc/gcc/install/lib64:install_dir/tools/
lib:install_dir/tools/lib/SuSE:${LD_LIBRARY_PATH}
```

For AIX, set your library path environment variable as follows:

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

For the 32-Bit Version

For Red Hat Linux, set your library path environment variable as follows:

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

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

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

For AIX, set your library path environment variable as follows:

```
setenv LIBPATH install_dir/tools/lib:${LIBPATH}
```

Running the Simulator in a Single Step

The `irun` command lets you run the simulator by specifying all input files and command-line options on a single command line. `irun` takes files from different simulation languages, such as Verilog, SystemVerilog, 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, `irun` automatically runs `ncelab` to elaborate the design and then `ncsim` to simulate the design. For more information, see the following topics:

- For details about how `irun` works, see the *irun User Guide*.
- For AMS-specific information, see Chapter 9, "Using irun for AMS Simulation."
- For information about using `irun` for design verification, see Chapter 10, "Using the AMS Designer Simulator for Design Verification."
- For information about using `irun` for designing with multiple power supplies, see Chapter 11, "Designing with Multiple Power Supplies."
- See also *Virtuoso® AMS Designer Simulator Tutorials*.

Running the Simulator in Three Steps

You can run the Virtuoso AMS Designer simulator in steps by running the three main executables in succession. Each executable has its own command-line syntax and arguments. See “Compiling the Design with ncvlog” on page 41, “Elaborating the Design with ncelab” on page 41, and “Simulating the Design with ncsim” on page 43 for more information.

You must have a library definition file (`cds.lib`) and you should have an `hdl.var` file. While these files can be very basic, they can become quite complex. See “Setting Up Your Design Environment for the Three-Step Approach” on page 40.

Cadence recommends using the three-step approach to run the AMS Designer simulator for designs that are organized in a library-based system. In contrast, some simulators, such as Verilog-XL, use a file-based system. You should also use the three-step approach if you do not depend on being able to switch between the Virtuoso AMS Designer simulator and Verilog-XL.

The three-step approach

- Provides more flexibility and control over the placement and reuse of intermediate files.
- Uses a simpler set of binding rules than the single-step approach.

  Binding is more predictable and manageable for the three-step approach. For more information about binding, see “Binding during Elaboration” on page 442.
- Provides finer control over the update mechanisms.
- Provides better incremental recompile performance for designs that continually rescan a directory or several directories or files.

  You can eliminate this behavior by organizing the design in a library-based system, making this process much more efficient.

See the following topics for more information:

- Setting Up Your Design Environment for the Three-Step Approach on page 40
- Compiling the Design with ncvlog on page 41
- Elaborating the Design with ncelab on page 41
- Simulating the Design with ncsim on page 43
- Understanding the Simulator Library Databases on page 45
- Using a Configuration on page 46

Setting Up Your Design Environment for the Three-Step Approach

Virtuoso® AMS Designer simulator libraries contain compiled objects (modules, macromodules, and user-defined primitives) and other derived data. The library structure consists of a Library.Cell:View (L.C:V) approach such that

- A library relates to a specific design or to a reference library
- Cells relate to specific modules or building blocks of the design
- Views relate to different representations of the building blocks

Each library has a logical name that corresponds to a unique directory. When you finish compiling and elaborating a design, the software creates a single file containing all of the internal representations of cells and views that the simulator requires and puts that file in the library directory.

To run the Virtuoso AMS Designer simulator, you need to set up a `cds.lib` file. This file contains statements that define your libraries and map logical library names to physical directory paths.

In addition, you can create an `hdl.var` file. This file defines which library is the work library and can contain definitions of other variables that configure your design environment, control the operation of Cadence® software, and specify the locations of support files and startup scripts.

Note: If you run the Virtuoso AMS Designer simulator using `irun`, the software creates `cds.lib` and `hdl.var` files for you automatically.

You can have more than one `cds.lib` or `hdl.var` file. By default, the AMS Designer simulator searches for these files in the following locations and uses only the first one it finds:

- The current work directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- `$HOME`
- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site-specific location, if defined)
- *`your_install_dir`*`/share`

You can create a `setup.loc` file to change the directories to search or to change the search order for the `cds.lib` and `hdl.var` files.

Compiling the Design with ncvlog

■ `ncvlog` compiles your source.

The compiler checks the syntax of the HDL design units (modules, macromodules, or user-defined primitives) in the input source files and generates an intermediate representation for each HDL design unit. The software stores these intermediate representations in a single file in the library directory and names the database file as follows:

inca.*architecture*.*lib_version*.pak

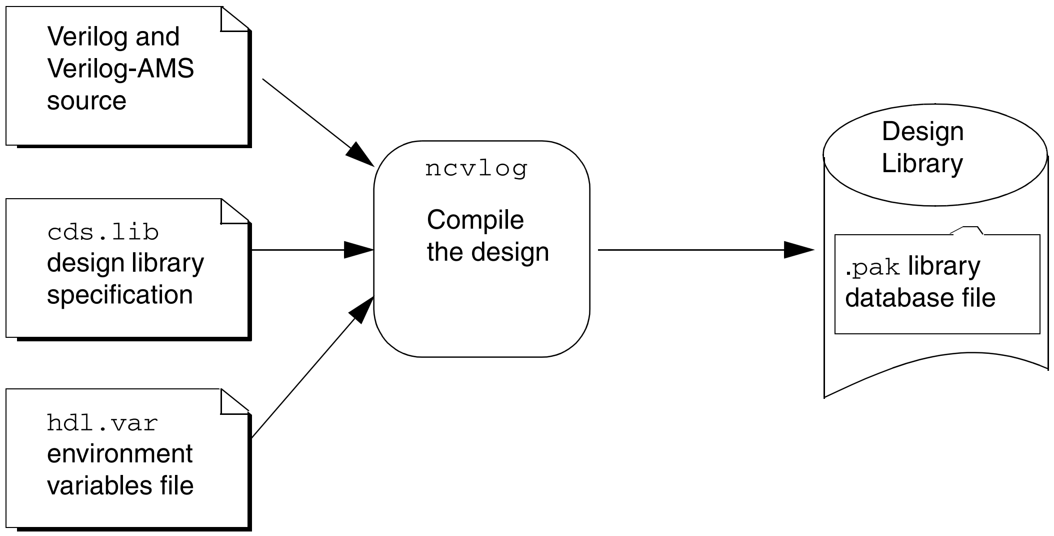
For example, the name of the library database file might be something like the following:

`inca.sun4v.091.pak`

See "Understanding the Simulator Library Databases" on page 45 for more information on library databases.

Using the single-step startup approach, `ncvlog` creates a binding list that the elaborator uses. Any change that you make to a source file causes the software to regenerate that binding list.

The following figure shows the inputs and outputs of `ncvlog`:


Verilog and Verilog-AMS source
cds.lib design library specification
hdl.var environment variables file
ncvlog
Compile the design
Design Library
.pak library database file

See Chapter 13, "Compiling," for more information about compiling with `ncvlog`.

Elaborating the Design with ncelab

■ `ncelab` elaborates the design.

The elaborator takes as input the Library.Cell:View name of the top-level HDL design unit. It then constructs a hierarchy based on the instantiation and configuration information in the design, establishes connectivity, and computes the initial values for all of the objects in the design.

If `ncelab` does not find any errors, it produces a snapshot. The snapshot, which contains the simulation data at simulation time 0, is the input to the `ncsim` simulator.

The software stores both the machine code and the snapshot in the library database file, along with the intermediate objects that are the result of compilation.

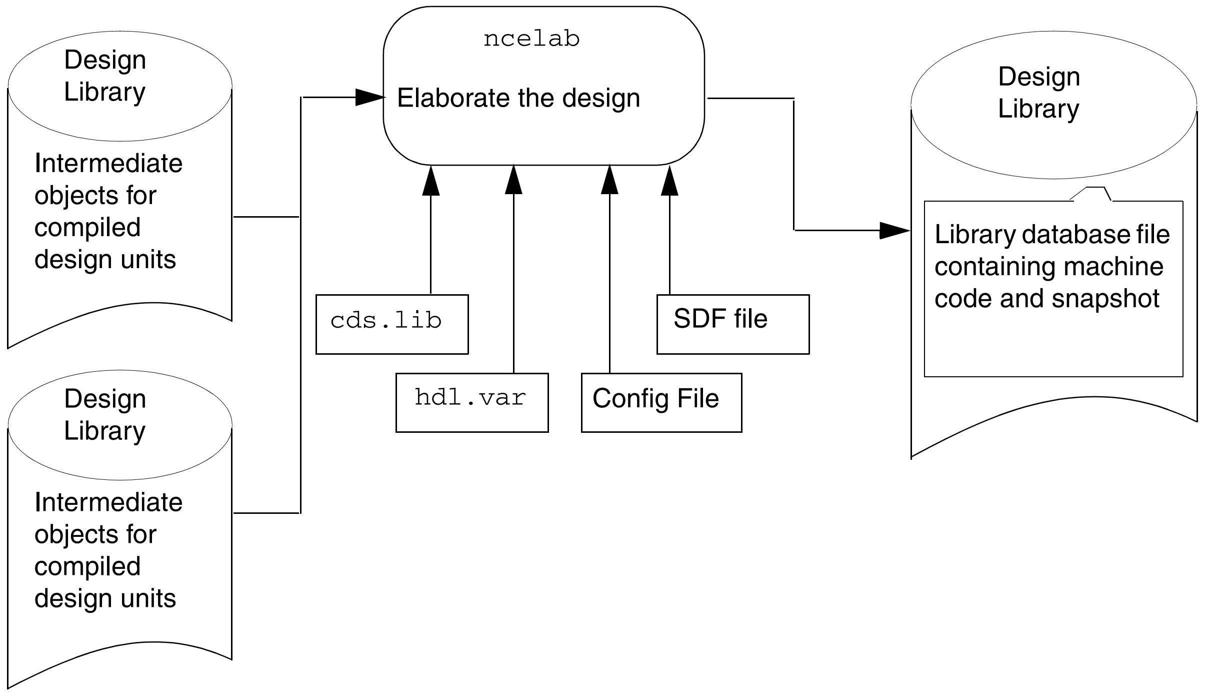
By default, the elaborator generates a snapshot that contains simulation constructs that have no read, write, or connectivity access. By limiting access to simulation objects, the elaborator can perform several optimizations that greatly increase performance.

When you run simulations in “regression” mode, the default access level is the obvious choice. However, if you run the simulator in this mode, you cannot access objects from a point outside the HDL code. For example, you cannot probe objects that do not have read access and the software cannot generate waveforms for these objects.

To run the simulation in debug mode with access to simulation objects, use the `-access` option to enable different kinds of access to simulation objects. You can also include an access file using the `-afile` option to specify the access capability for particular instances and for parts of a design.

See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 443 for more information on running the Virtuoso AMS Designer simulator in regression mode versus running the simulator in debug mode.

The following figure shows the inputs and outputs of `ncelab`:


ncelab
Elaborate the design
Design Library
Intermediate objects for compiled design units
Design Library
Intermediate objects for compiled design units
cds.lib
hdl.var
Config File
SDF file
Design Library
Library database file containing machine code and snapshot

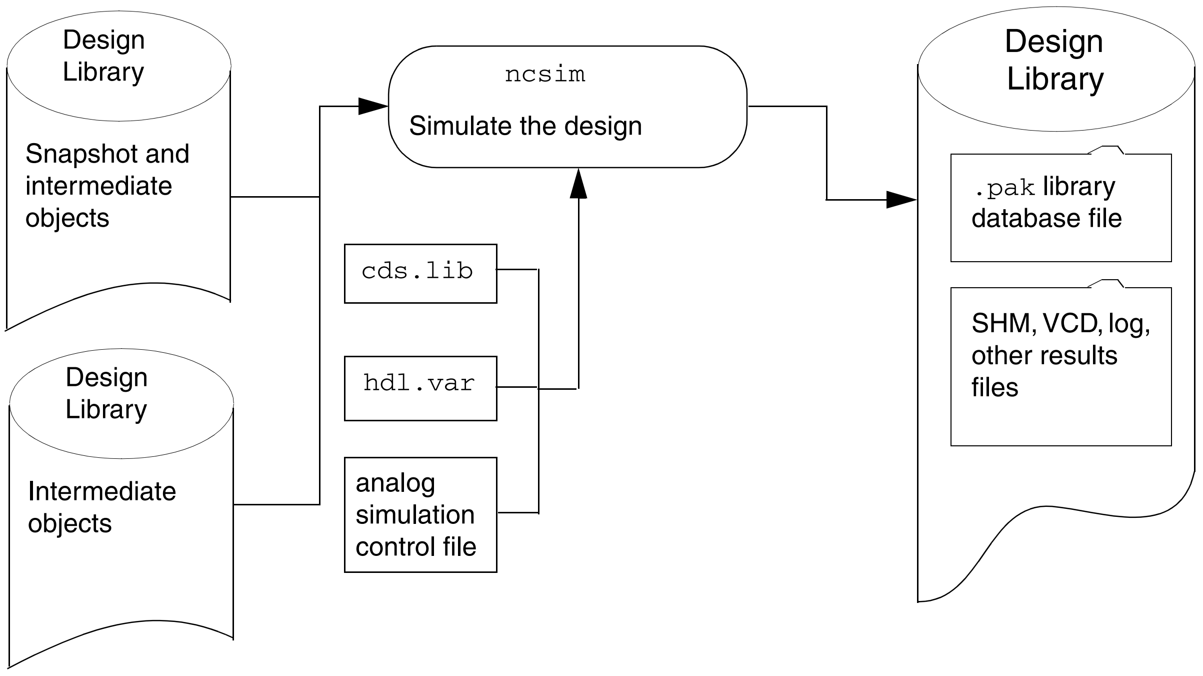
Using the multi-step approach, the elaborator makes all binding decisions. Using the single-step approach, the elaborator uses the binding list that `ncvlog` generates.

See Chapter 14, "Elaborating," for more information about elaborating with `ncelab`.

Simulating the Design with ncsim

The simulator loads the snapshot that the elaborator generated, as well as other objects the compiler and elaborator generate that the snapshot references. The simulator might also load HDL source files, script files, and other data files (using `$read*` tasks or `textio`). `ncsim` can generate a log file, an SHM or VCD database, and other results files.

The following figure shows the inputs and outputs of `ncsim`.


Design Library
Snapshot and intermediate objects
Design Library
Intermediate objects
ncsim
Simulate the design
cds.lib
hdl.var
analog simulation control file
Design Library
.pak library database file
SHM, VCD, log, other results files

See Chapter 15, "Simulating," for more information.

You can run the `ncsim` simulator

- In noninteractive mode: The simulation runs immediately after initialization
- In interactive mode: The simulator stops to await input before simulating time zero

You can also run the simulator with the Cadence SimVision environment. The SimVision environment is a comprehensive debug environment that consists of

- A main window in which you can view your source and perform a wide variety of debug operations
- Advanced debug programs that you can access from the main window:
  - The Navigator lets you view your current design hierarchy in a graphical tree representation and as a list of objects with their current simulation values and declarations
  - The Watch Window lets you select and then watch signal value changes
  - The Trace Signals sidebar lets you trace backwards through a design from a signal that has a questionable value to where a signal first diverges from the expected behavior

The SimVision environment also includes the SimVision waveform viewer and SimCompare, an application that lets you compare results stored in SHM (SST2) or VCD databases. See the *Introduction to SimVision* and the *Using SimCompare* books for more information.

Because the Virtuoso AMS Designer simulator is a compiled code simulator that does not contain an interpreter, and because `ncsim` must be able to display and manipulate mixed-language constructs, you cannot type Verilog or Verilog-AMS commands at the command-line prompt. Instead, the AMS Designer simulator supports a set of Tool Command Language (Tcl) commands for interactive debugging. See Appendix B, "Tcl-Based Debugging," for a list of interactive commands.

Note: If you run `ncelab` in the default (regression) mode to elaborate the design, you will not have read, write, or connectivity access to simulation objects. A warning or error message appears if you execute a Tcl command that requires read or write access. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for more information.

You can use Tk with the AMS Designer simulator. Tk is a toolkit for the X Windows System that extends the Tcl facilities with commands that you can use to build user interfaces, so that you can develop Motif-like user interfaces by writing Tcl scripts instead of writing C code. Cadence does not ship Tk with the simulator. However, the required shared library and the library of Tcl script files is available on the internet. See *Incisive Simulator Tcl Command Reference* for instructions on enabling Tk in the Cadence NC-Verilog simulator.

Understanding the Simulator Library Databases

When you compile and elaborate a design, the software stores all intermediate objects in a single file in the design library. This library database file is called

`inca.`*`architecture.lib_version`*`.pak`

For example, the name of the library database file might look like the following:

```
inca.sun4v.091.pak
```

Library database files by default have both read and write access. You can use the `ncpack` utility to change the properties of a database to make it read-only or add-only.

The software uses a file-locking mechanism to manage multiple processes that might need to read or modify the contents of a library at one time. If a process cannot get a required lock, the AMS Designer simulator issues a warning and the process tries again a short time later. If a process cannot get a lock after approximately one hour, the process times out and exits.

Here are two examples of warning messages that the file-locking mechanism might issue:

```
ncvlog: *W,DLWTLK: Waiting for a read lock on library 'alt_max2'.
ncvhdl_cg: *W,DLWTLK: Waiting for a write lock on library 'worklib'.
```

In rare cases, file locking results in a deadlock in which neither process can proceed because each is waiting for the other to release a lock. For example, some processes that you suspend with a `Control-Z` retain their locks (`ncelab` is such a process). In these cases, you must use the `ncpack -unlock` command to terminate the process.

A signal-handling mechanism ensures that when an unexpected event, such as a `Control-C`, occurs the simulator flushes the database to the disk. However, terminating a process with `kill -9` or a power failure can corrupt a library database. In these cases, you must delete the library database file and rebuild.

The following example shows a message that indicates a corrupted library:

```
ncvlog: *F,DLPAKC: Packed library for alt_max2 is corrupt,
    please remove ./alt_max2/inca.sun4v.091.pak.
```

Using a Configuration

A configuration is a set of rules that specifies which cellviews under a top-level cell are part of a design for a given purpose (such as elaboration or simulation). The cellview for the top-level cell contains the configuration. You can use the hierarchy editor (HED) to create configurations (see the *Cadence Hierarchy Editor User Guide* for more information).

To use configurations in the Virtuoso AMS Designer flow, follow these guidelines:

- Compile the design with the `-use5x` command-line option and make sure the design is located in a Cadence library.

  For more information, see "ncvlog Command Syntax and Options" on page 397.

- Use the `-use5x4vhdl` command-line option when you elaborate the design.

  This option applies configurations to VHDL as well as Verilog-AMS modules.

- (Optional) Use `ncelab -snapshot` to specify a different location for the simulation snapshot (other than the cellview directory of the first design unit specified on the `ncelab` command line).

Checking Analog Solver-Related Information

The AMS simulation log prints the MMSIM version of the analog solver used with the AMS Designer simulator. This log contains the same type of messages that are printed in the MMSIM simulator log. If you need to determine the precise MMSIM version without running AMS simulation, type the following at the command prompt:

```
print_mmsimver
```

This command returns the exact MMSIM version number, as is returned by the `spectre -W` command in an MMSIM installation. For example:

```
MMSIM Version: 7.1.0.060.isr4
```

You can also use the following `ncsim` commands to display the exact MMSIM ISR version number, without having to run simulation.

```
ncsim -spectre_args -W
ncsim -spectre_args -V
```

Note: The MMSIM version printed out by the above commands applies to all MMSIM products of Spectre, UltraSim, and APS that are integrated with AMS Designer.

In addition, the ncsim simulation log contains the MMSIM 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 MMSIM products in MMSIM releases.

```
Cadence (R) Virtuoso (R) Spectre (R) Circuit Simulator
Version 7.1.0.034.isr15 -- 2 Mar 2009
```

The ncsim log also includes a short summary of the analog solver used with the AMS Designer simulator. The log also prints the arguments used with the solver, if any. For example:

```
AMSD: Using aps solver with arguments: +parasitics=10 +mt=4.
```

Using Advanced Analog Solver Features

The AMS Designer simulator supports the Spectre, APS, and UltraSim simulator solvers. You can also access some of the advanced features of the Spectre XL, Spectre GXL, and

UltraSim XL tiered products. The following tables outline the XL and GXL features available when you use the advanced solvers with the AMS Designer simulator.

Spectre XL Feature	**AMS Designer + Spectre Solver**
Spectre RF	See “Envelope Analysis (envlp)” on page 77

Spectre GXL Feature	**AMS Designer + Spectre Solver**
Parasitic reduction	See “Turning On Spectre Parasitic Reduction” on page 304

UltraSim XL Feature	**AMS Designer + UltraSim Solver**
RC reduction (`postl`)	Use
Stitching/backannotation	For SPICE netlist only
Voltage regulator (`.usim_vr`)	Use
Power network solver (`.usim_pn`, `.usim_ups`)	Not available; iteration=1 only
Dynamic checks (`.acheck`, `.dcheck`, `.pcheck`)	`.dcheck` and `.pcheck` only
Static checks (`.usim_report partition`, `.usim_report node`, `.usim_report chk_param`, `.usim_report chk_sustrate`)	`.usim_report partition` and `.usim_report node` only
Flash model (`.appendmodel`)	See `.appendmodel` in “UltraSim Solver Control Statements” on page 85
EMIR (`.usim_emir`)	Not available

More about UltraSim Features

In addition to the above features of the UltraSim solver, if you are running a simulation in `sim_mode=a`, the UltraSim-Turbo feature is turned on by default. This allows the AMS Designer simulator to leverage the performance advantage of the UltraSim Turbo engine. However, the Turbo performance enhancement does not apply to any mode other than `sim_mode=a`.

For more information about UltraSim, see the *Virtuoso UltraSim Simulator User Guide*.

3 Using the 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 `ncsim` (or to the simulation phase of `irun`). See "Specifying Controls for the Analog Solvers" on page 57 for information about the statements you can use in the analog simulation control file.

See also the following topics:

- Using Spectre Language Statements in the Analog Simulation Control File on page 52
- Using UltraSim Statements in the Analog Simulation Control File on page 52
- Switching Languages in the Analog Simulation Control File on page 54
- Passing the Analog Simulation Control File to ncsim or irun on page 55

Using 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. In the remainder of the file, you can use both Spectre and UltraSim control statements. 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.

Note: 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
```

See "Specifying Controls for the Analog Solvers" on page 57 for more information.

Using UltraSim Statements in the Analog Simulation Control File

You cannot have active UltraSim statements in an analog simulation control file that is meant for the Spectre solver. One way to handle this restriction is to comment out lines that do not apply to the solver you are using.

Alternatively, if you write your UltraSim control statements using the following format, you can use the same analog simulation control file for both the UltraSim and Spectre solvers: The UltraSim solver recognizes and acts upon statements of this form while the Spectre solver ignores them without warning.

```
*ultrasim: UltraSim_statement
```

For example, consider the two additional statements at the end of the following analog simulation control file. Both the Spectre and the UltraSim solvers recognize the `save` statement. The last statement is an UltraSim statement placed in a comment so that you can use this file with the Spectre solver, too.

```
//Control File 5c -- The first line is always a comment the slashes are optional
saveNodes options rawfmt=sst2
```

```
timeDom tran stop=5n
save top.a.b.c
*ultrasim: .probe tran v(*) depth=2
```

If you write the last line without the `*ultrasim:` prefix,

```
.probe tran v(*) depth=2
```

you cannot use the file with the Spectre solver.

See also

- UltraSim Solver Control Statements on page 85 for information about control statements you can use with the UltraSim solver
- Switching Languages in the Analog Simulation Control File on page 54 for some restrictions that apply to using control statements with the UltraSim solver

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 by typing the following statement at that point in the file:

```
simulator lang=spice
```

You cannot use Spectre language statements in a SPICE language section of the control file, but you can use the uncommented form of UltraSim statements. When you set the simulator language to SPICE, the simulator uses UltraSim look-up rules: The simulator is insensitive to the case of node and hierarchical names, keywords, parameters, units, and the contents of `.vec` and `.vcd`/`.evcd` files, especially the node names.

You can force the software to use Spectre lookup rules so that the simulator is sensitive to the case of node and hierarchical names but insensitive to the case of keywords, parameters, and units by specifying the `lookup` option on the `simulator lang` line as follows:

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

You must specify `lookup=spectre` when you use `.meas`, `.probe`, and `.print` SPICE commands and you must use the full hierarchical node name in these commands. For example, if you have a top-level module, `myTop`, in which there is a node, `myNode`, you specify your `.meas` statement in the analog simulation control file as follows:

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

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

```
simulator lang=spectre
```

In the analog simulation control file, you can use the Spectre language statements described in the “Control Statements” chapter of the *Virtuoso Spectre Circuit Simulator User Guide*. You can also use the UltraSim simulation and control statements described in the *Virtuoso UltraSim Simulator User Guide*. The following exceptions apply:

- The UltraSim solver does not support `options save`. Instead, use `.probe`.
- You can use the `save` statement with both the Spectre and UltraSim solvers.

Passing the Analog Simulation Control File to ncsim or irun

You can specify the analog control file directly on the `irun` command line. For example, to pass an analog control file called `adc.scs` to `irun` to compile, elaborate, and simulate a design contained in a file called `sar6bit.v`, type the following:

```
irun adc.scs sar6bit.v
```

To pass the analog simulation control file to `ncsim`, you must use the `-analogcontrol` command-line option:

```
ncsim -analogcontrol analogControlFileName …
```

So, to pass an analog control file called `adc.scs` to `ncsim` to compile, elaborate, and simulate a design contained in a file called `sar6bit.v`, type the following:

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

4

Specifying Controls for the Analog Solvers

In the Virtuoso® AMS Designer simulator, the primary way of controlling the analog solvers is to define an analog simulation control file. You can read about the statements you can use in the analog simulation control file in the following sections:

See also

- Mixed-Signal DC Initialization on page 98
- Time-Saving Techniques for the Analog Solvers on page 100

Language Mode (simulator lang)

The `simulator lang` statement specifies the language mode you want the software to use to evaluate succeeding statements. See the following topics for more information:

- Using Spectre Language Statements in the Analog Simulation Control File on page 52
- Using UltraSim Statements in the Analog Simulation Control File on page 52
- Switching Languages in the Analog Simulation Control File on page 54

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 *Virtuoso 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.
- 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. The UltraSim solver supports a subset of these parameters as specifically noted in the **Definition** column.

Parameter	**Definition**
`approx=no`	Uses approximate models. Difference between approximate and exact models is generally very small. Values: `no`, `yes`
`audit=detailed`	Print time required by various parts of the simulator. Values: `no`, `brief`, `detailed`, `full`
`compatible=spectre`	Encourage device equations to be compatible with a foreign simulator. This option does not affect input syntax. Values: `spectre`, `spice2`, `spice3`, `cdsspice`, `hspice`, `spiceplus`
`currents=selected`	Terminal currents to output. Values: `all`, `nonlinear`, `selected`
`debug=no`	Give debugging messages. Values: `no`, `yes`
`diagnose=no`	Print additional information that might help diagnose accuracy and convergence problems. Values: `no`, `yes`
`digits`	Number of digits used when printing numbers.
`error=yes`	Give error messages. Values: `no`, `yes`
`gmin=1e-12 S`	Minimum conductance across each nonlinear device.
`gmin_check=max_v_only`	Specifies that effect of `gmin` should be reported if significant. Values: `no`, `max_v_only`, `max_only`, `all`
`homotopy=all`	Method used when no convergence on initial attempt of DC analysis. Values: `none`, `gmin`, `source`, `dptran`, `ptran`, `all`
`iabstol=1e-12 A`	Current absolute tolerance convergence criterion.
`ignshorts=no`	Silently ignore shorted components. Values: `no`, `yes`
`info=yes`	Give informational messages. Values: `no`, `yes`

Parameter	Definition
`inventory=detailed`	Print summary of components used. Values: `no`, `brief`, `detailed`
`limit=dev`	Limiting algorithms to aid DC convergence. Values: `delta`, `log`, `dev`
`macromodels`	Circuit contains macromodels. Providing this information sometimes helps performance.
`maxnotes=5`	Maximum number of times any notice will be issued per analysis.
`maxnotestologfile`	Maximum number of times any notice will be printed to the log file per analysis.
`maxrsd=0`	Threshold below which parasitic node reduction occurs.
`maxwarns=5`	Maximum number of times any warning message is issued per analysis.
`maxwarnstologfile`	Maximum number of times any warning message is printed to the log file per analysis.
`minr=0.0`	Threshold below which resistance inside devices is ignored.
`mos_method=s`	Method used to evaluate BSIM3V3 and BSIM4 models. Values: `a` (for accelerated, using table models when available), `s` (for standard, using standard analytical evaluation) If this option is set to `a` and the corresponding option on a BSIM model card is set to `a` (the default), the simulator uses table models to simulate the bsim model. If this option is set to `a` but the corresponding option on a BSIM model card is set to `s`, the simulator uses standard analytical evaluation for that bsim model. For more information, see the "Analyses" chapter in the *Virtuoso Spectre Circuit Simulator User Guide*.
`mos_vres=0.05`	Voltage increment for the MOSFET table model interpolation grid.
`narrate=yes`	Narrate the simulation. Values: `no`, `yes`
`notation`	Notation to be used when printing real numbers to the screen.

Parameter	Definition
`note=yes`	Give notice messages. Values: `no`, `yes`
`opptcheck=yes`	Check operating point parameters against soft limits. Values: `no`, `yes`
`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: `no`, `yes`
`pivrel=0.001`	Relative pivot threshold.
`quantities=no`	Print quantities. Values: `no`, `min`, `full`
`rawfile="%C:r.raw"`	Output raw data file name, optionally including an absolute or relative path.
`rawfmt=sst2`	Output raw data file format. Values: `psfbin`, `psfascii`, `fsdb`, `wdf`, `sst2`, `psfxl` The specified format affects only analog signals. For AC analysis, the only supported values are `psfbin` and `psfascii`. Only signals that you select by using Tcl commands prior to running the AC analysis are saved. `fsdb` (Fast Signal Database) and `wdf` 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 `sst2`. 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 `psfbin`, `psfascii`, `fsdb`, `wdf`, `sst2`, and `psfxl`.

Parameter	Definition
`redundant_currents=no`	If yes, save both currents through two terminal devices. Values: `no`, `yes`
`reltol=0.001`	Relative convergence criterion.
`rforce=1`	Resistance used when forcing nodesets and node-based initial conditions.
`save=selected`	Signals to output. Values: `all`, `allpub`, `selected`, `none`
`scale=1`	Device instance scaling factor. The UltraSim solver also supports this option.
`scalem=1`	Model scaling factor. The UltraSim solver also supports this option.
`sensfileonly=no`	Enable or disable raw output of sensitivity results. Values: `no`, `yes`
`speed`	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`. **Note:** You can set this parameter for the Spectre and APS solvers only, but a similar capability (with eight possible values) is available for the UltraSim solver using the `.usim_opt` parameter.
`temp=27 C`	Temperature. The UltraSim solver also supports this option.
`tempeffects=all`	Temperature effect selector. Values: `vt`, `tc`, `all`
`tnom=27 C`	Default component parameter measurement temperature. The UltraSim solver also supports this option.
`topcheck=full`	Check circuit topology for errors. Values: `no`, `min`, `full`

Parameter	Definition
`useprobes=no`	Use current probes when measuring terminal currents. Values: `no`, `yes` **Note:** The following devices always use probes to save currents (even with `useprobes=no`): `port`, `delay`, `switch`, `hbt`, `transformer`, `core`, `winding`, `fourier`, `d2a`, `a2d`, `a2ao`, `a2ai`. You cannot use this parameter to save currents during an AC analysis.
`vabstol=1e-06 V`	Voltage absolute tolerance convergence criterion.
`warn=yes`	Give warning messages. Values: `no`, `yes`

AC Analysis (ac)

Important

Only the Spectre solver supports AC analysis. The UltraSim solver ignores `ac` statements. 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 }
```

For information about specifying an AC analysis, see “AC Analysis” in the *Virtuoso Spectre Circuit Simulator User Guide*. See also “Mixed-Signal DC Initialization” on page 98.

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.

Note: During AC analysis, you *cannot* probe currents or digital signals.

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.

Parameter	Definition
`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 `probe` 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.

Transient Analysis (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, at most, one transient analysis in your analog simulation control file.

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

For more information, see "Transient Analysis" in the *Virtuoso Spectre Circuit Simulator User Guide*. See also the *Virtuoso UltraSim Simulator User Guide*.

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:	
	, `sweep`, `status`, and `steps` all	
	`no`	Suppresses information about the tran analysis

	`title`
	`sweep`
	`status`
	`steps`
		Produce annotation information such as the number of transient analysis steps and how long the analysis takes

`cmin=0 F`	Minimum capacitance from each node to ground.	

Parameter	Definition
`compression=yes`	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: `yes`, `no`
`compfactor=1.0`	Compression factor. Limits the tolerance values during compressing transient waveforms. The compression decision is made according to `tolerance * compfactor`.
`errpreset=moderate`	Selects a reasonable collection of parameter settings. Values: `conservative`, `moderate`, `liberal`
`fastbreak=no`	Specifies the evaluation method to use for VHDL-AMS `break` 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 **Note:** 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.
`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. You can also use this parameter with the UltraSim solver. Values:

Parameter	Definition	
	`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 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, or when you use the UltraSim solver, the default is `cm`.	
`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. You can also use this parameter with the UltraSim solver.	

Parameter	Definition
`infotimes=[...] s`	Times when the simulator is to perform the `info` analysis specified by the `infoname` parameter. You can also use this parameter with the UltraSim solver. 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. ■ The UltraSim solver does not do the analysis at all.
`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`. **Note:** If you are using the UltraSim solver, use the `.usim_opt` keyword to specify a `maxstep` value.
`method`	Integration method. Default derived from `errpreset`. Values: `euler`, `trap`, `traponly`, `gear2`, `gear2only`, `trapgear2` The UltraSim solver ignores the `trapgear2` value. If you specify `trapgear2`, the UltraSim solver uses a default value that depends on the `sim_mode` parameter or uses the method you specify using the `.usim_opt method` parameter, if available.
`oppoint=no`	Should operating point information be computed for initial timestep, and if so, where should it be sent. Values: `no`, `screen`, `logfile`, `rawfile` This parameter is not supported by the UltraSim solver. However, you can obtain similar information by using the `infotimes` parameter with a time of zero.
`outputstart=`*`time`*	Output is saved only after this time is reached. You can use this parameter *only* with the UltraSim solver.

Parameter	Definition
`readic`	File that contains the initial conditions. You can also use this parameter with the UltraSim solver.
`readns`	File that contains the estimate of initial transient solution. You can also use this parameter with the UltraSim solver.
`relref`	Reference used for the relative convergence criteria. Default derived from `errpreset`. Values: `pointlocal`, `alllocal`, `sigglobal`, `allglobal`
`save`	Signals to output. Values: `all`, `selected`, `none` *Important* Instead of using this `save` parameter to save signals, Cadence recommends using the Tcl `probe` command, which has fewer restrictions.
`skipcount`	Save only one of every skipcount points.
`skipdc=no`	If `yes`, 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 `readic` parameter. If the `readic` parameter is not specified, the values specified on the `ic` statements are considered. Device-based initial conditions are not used for `skipdc`. Nodes that you do not specify with the `ic` file or `ic` 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: `no`, `yes`, `waveless`, `rampup`, `autodc` *Caution* ***Using the AMS Designer simulator, setting*** `skipdc` ***to anything other than*** `no` ***can cause incorrect signal transitions between analog and digital during transient analysis.***
`skipstart=`*`starttime`* `s`	

Parameter	Definition
	The time to start skipping output data.
`start=0 s`	Start time.
`stats=no`	Analysis statistics. Values: `no`, `yes`
`step=0.001 (`*`stop-start`*`) s`	Minimum time step used by the simulator solely to maintain the aesthetics of the computed waveforms. You can also use this parameter with the UltraSim solver.
`stop (s)`	Stop time. You can also use this parameter with the UltraSim solver.
`strobedelay=0 s`	The delay (phase shift) between the skipstart time and the first strobe point. You can also use this parameter with the UltraSim solver.
`strobeperiod (s)`	The output strobe interval (in seconds of transient time). You can also use this parameter with the UltraSim solver.
`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.

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.

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.

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

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

`noisefmax=0 (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/noisefmax.
`noiseon=[...]`	The list of instances to be considered as noisy during transient noise analysis.
`noiseoff=[...]`	The list of instances to be considered as not noisy during transient noise analysis.
`noisescale=1`	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.
`noiseseed`	Seed for the random number generator. Specifying the same seed allows you to reproduce a previous experiment.

`noisefmin (Hz)`
: If specified, the power spectral density of noise sources depend on frequency in the interval from `noisefmin` to `noisefmax`. Below `noisefmin`, noise power density is constant. The default value is `noisefmax`, so that only white noise is included and noise sources are evaluated at `noisefmax` for all models. 1/`noisefmin` cannot exceed the requested time duration of transient analysis.

`noisetmin (s)`
: Minimum time interval between noise source updates. Default is 1/`noisefmax`. Smaller values will produce smoother noise signals at the expense of reducing time integration step.

`noiseupdate=fmax` | `step`
: Specifies whether noise is to be injected at a constant time step (`fmax`) or the Spectre solver time step is to be used (`step`). Injecting noise at a constant time step is suitable when the value of `noisefmax` is larger that 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 `noisefmax`, which forces the simulator to take steps smaller than `noisetmin`, 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
```

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 (`ncsim`) simulation flow of multiple simulation runs under parameter variations in a single `ncsim` invocation.

The use model of Monte Carlo analysis in AMS Designer is the same as that in Spectre. For more information, refer to the “Monte Carlo Analysis” section in the “Control Statements” chapter of the *Virtuoso Spectre Circuit Simulator User Guide*.

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 `irun` or `ncsim` 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.

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

Initial Conditions (ic)

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
node=*value*	*Important* For AMS simulation, you must always include the top cell name in the *node* specification. Each *node* 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.

See also:

- “Initial Conditions (ic)” in the “Syntax” chapter of the *Virtuoso Spectre Circuit Simulator Reference*
- ".ic" in “Simulation and Control Statements” in the "Netlist Formats" chapter of the *Virtuoso UltraSim Simulator User Guide*

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.

  Note: All three solvers (Spectre, UltraSim, and APS) support ***`node`*****`=`*****`value`*** syntax as well as `v(`*`node`*`)=`*`value`* syntax.

Initial Guess (nodeset)

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.

Important

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

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

For more information, see “Node Sets (nodeset)”, in the “Spectre Syntax” chapter of *Virtuoso Spectre Circuit Simulator Reference*.

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:t1=1.0u
```

specifies that

- Node `n1` should be about 0 V,
- Node `out` 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.

Envelope Analysis (envlp)

The AMS Designer simulator supports the Spectre RF envelope analysis (`envlp`). The signal sources must be Spectre primitives.

See

- "Envelope (envlp) Choosing Analyses Form" in the *Virtuoso Spectre RF Simulation Option User Guide* for information about setting up `envlp` analysis using the Virtuoso Analog Design Environment with the Spectre RF option
- "Envelope Analysis" in the *Virtuoso Spectre RF Simulation Option Theory* for detailed information on `envlp` analysis and its settings

The Spectre RF envelope solver detects analog-to-digital (A-to-D) or digital-to-analog (D-to-A) events, skipping as many high-frequency cycles as possible at each envelope step. The AMS Designer simulator synchronizes the digital and analog simulations in an interval around time points where the A-to-D or D-to-A events occur.

You can specify the following parameters for AMS-`envlp` cosimulation in addition to those documented in the "ENVLP Parameters" section of "Envelope Analysis" in the *Virtuoso Spectre RF Simulation Option Theory*:

Parameter	Valid Values	Description
`resetenv`	`yes`	Resets envelope data after D-to-A or A-to-D events for AMS-`envlp` cosimulation
	`no`	Does not reset envelope data **Note:** This is the default value.
`ignoredclk`	`yes`	Tells the simulator to ignore the digital clock if the clock rate is in the same order as the envelope clock for AMS-`envlp` cosimulation
	`no`	Do not ignore the digital clock **Note:** This is the default value.
`trancycles`	*`x_numCycles`*	Specifies an integer number of transient cycles around time points for D-to-A or A-to-D events for AMS-`envlp` cosimulation **Note:** The default value is 5.

Note: Envelope analysis does not support behavioral module components with hidden states.

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 (ADE) GUI.

For more information, refer to “The assert Statement” section in the “Control Statements” chapter of the *Virtuoso Spectre Circuit Simulator User Guide*.

Displaying and Saving Information (info)

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. (See also "Mixed-Signal DC Initialization" on page 98.) 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
`all`	Outputs a list of input and output parameter values.
`input`	Outputs information about inputs. Also supported by the UltraSim solver.
`inst`	Outputs information about instances. Also supported by the UltraSim solver.
`models`	Outputs information about models. (This setting is effective only for Verilog®-AMS.) Also supported by the UltraSim solver.
`nodes`	Outputs a terminal-to-node map.
`none`	Does not print any parameters.

Settings	Action
`oppoint`	Outputs device parameter information about DC operating points. This is the default. Also supported by the UltraSim solver. To request node voltage output during an operating point analysis, specify the `info writedc` parameter.
`output`	Outputs information about outputs. (This setting is effective only for Verilog-AMS.) Also supported by the UltraSim solver.
`parameters`	Outputs netlist parameter values such as `temp` and `tnom`. (This setting is effective only for Verilog-AMS.)
`subckts`	Outputs subcircuit parameters.
`terminals`	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, the APS solver, or the UltraSim solver:

Settings	Action
`file`	Sends the output to a file that you create. If you use this setting, use the `file` parameter to specify the output file.
`logfile`	Sends the parameters to the simulator log file. This is the default.
`nowhere`	Produces no output.
`rawfile`	Sends the output to rawfile in the format specified by the `options` statement `rawfmt` parameter. The rawfile is `.raw/`*`myinfo`*`.info` where *`myinfo`* is the name of the `info` statement.
`screen`	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
"*filename*"	Specifies a file to receive the output requested by the `info` statement. This is the file that is used when you specify the `where=file` parameter and setting. The default value is *scs*.info.*what_setting* where *scs* is the basename of the analog simulation control file that specifies the command, and *what_setting* is the active value for the `info -what` 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
no	Does not generate a summary of minimum and maximum parameter values.
only	Generates a summary of only minimum and maximum values.
yes	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**
`all`	Saves all signals.
`selected`	Saves only signals specified with `save` statements.
`none`	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 *`filename`*`.dc`.

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.

Specifying Signals to Save (save)

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.

Note: 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.i1`.

```
save top.i1.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.

Specifying Signals to Print (print)

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 *Virtuoso Spectre Circuit Simulator User Guide*.

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

UltraSim Solver Control Statements

The UltraSim solver control statements help specify the behavior of the UltraSim solver.

`*ultrasim:` *`keyword { option } { scope }`*

The asterisk causes the Spectre solver and the APS solver to ignore these statements, making it possible to use the same analog simulation control file for all three solvers. You must put a space between the colon and the initial dot (`.`) of the *`keyword`*.

The following keywords are among those you can use in UltraSim solver control statements. For additional keywords and syntax information, see the cross-references in the following table.

Keyword	Definition	Cross-Reference(s)
`.appendmodel`	Includes a MOSFET model card in a FLASH core cell model.	*Virtuoso UltraSim Simulator User Guide*: “Flash Core Cell Models”
.dcheck	Allows you to monitor metal oxide semiconductor (MOS) voltages during a simulation run and generates a report if the voltages exceed the specified upper or lower bounds, or meet the specified conditions.	*Virtuoso UltraSim Simulator User Guide*: “MOS Voltage Check” in “Virtuoso UltraSim Advanced Analysis”
`.pcheck` *`title`* `zstate...`	Specifies a high-impedance node check. **Note:** The AMS Designer simulator considers the analog port of interface elements to have a low-impedance path to ground, internally.	*Virtuoso UltraSim Simulator User Guide*: “High Impedance Node Check” in “Virtuoso UltraSim Advanced Analysis”
`.probe`	Sets up probes on nodes, ports, or elements for a specified output quantity.	“Details about the .probe Statement” on page 87
`.usim_nact`	Sets up a node activity analysis on the nodes in a circuit.	*Virtuoso UltraSim Simulator User Guide*: “Virtuoso UltraSim Advanced Analysis”

Keyword	Definition	Cross-Reference(s)

`.usim_opt`	Allows you to set various options for the UltraSim solver.
	
	Note: The software automatically adds the following options just prior to running the simulation:
	
	`.usim_opt del_allnode_inst=no`
	`.usim_opt wf_output_format=verilog`
	`.usim_opt wf_param_hier=1`
	`.usim_opt addflowsuffix=yes`
		“Details about the .usim_opt Options the Software Adds Automatically” on page 97
		
		See also the *Virtuoso UltraSim Simulator User Guide*: “Setting Virtuoso UltraSim Simulator Options”

`.usim_pa`	Sets up a power analysis on specified subcircuits.	*Virtuoso UltraSim Simulator User Guide*: “Virtuoso UltraSim Advanced Analysis”
`.usim_report node`	Reports nodes with an element connection larger than a threshold value you specify.	
`.usim_report partition`	Reports partition information for the 10 largest partitions.	
`.usim_ta`	Reports setup timing errors on specified nodes with respect to a reference node.	
`.usim_vr`	Sets up a voltage regulator simulation.	*Virtuoso UltraSim Simulator User Guide*: “Voltage Regulator Simulation”
`.vcd`	Enables the UltraSim solver to use a Verilog value change dump (VCD) file.	*Virtuoso UltraSim Simulator User Guide*: “Processing the Value Change Dump File”
`.vec`	Enables the UltraSim solver to process digital vector files.	*Virtuoso UltraSim Simulator User Guide*: “Digital Vector File Format”

Details about the .probe Statement

See the following topics for details about the UltraSim `.probe` statement:

- Syntax for .probe Statement on page 87
- Description of domain Switch Values on page 90
- Description for .probe Statement on page 91
- Examples for .probe Statement on page 94

Syntax for .probe Statement

```
probe_statement::=
    .probe tran [include_rc] {[output_name = ]output_var} [depth = value]
   [subckt = name] [exclude = pn1, pn2] [preserve = none|all|port]
   [domain = analog|digital|mixed] [vcd = yes|no]

output_var ::=
        v(node_name)
      | i(element_name)
      | x(instance_port_name)
      | x0(instance_port_name)
      | par('expression')
      | v(node1, node2)
```

.probe Option	Description
`tran`	The only supported analysis type: transient.
`include_rc`	Used with the `.probe v(*)` statement, which has a wildcard to match multiple nodes in a subcircuit. `include_rc` causes `V(*)` to match all nodes, including internal RC nodes. By default, internal nodes that are only connected to RC are not covered by `V(*)`.
`v(node_name)`	Probes the *`node_name`* voltage. The *`node_name`* can be hierarchical, and can contain question marks and wildcards. For example: `v(x?1.*.n*)`
`i(element_name)`	Prints the branch current output through the element *`element_name`*. The *`element_name`* can be hierarchical, and can contain question marks and wildcards. For example: `i(x?1.*.n*)`

.probe Option	Description
`x`*`(instance_port_name)`*	Returns the current flowing into the subcircuit port, including all lower hierarchical subcircuit ports. It can be used to probe power and ground ports of an instance, even if the ports are defined as a global node, and do not appear in the subcircuit port list. The *`instance_port_name`* can be hierarchical, and can contain question marks and wildcards. For example: `x(x?1.*.n*.vdd)`
`x0`*`(instance_port_name)`*	Returns the current flowing into the subcircuit port, excluding all other lower hierarchical subcircuit ports. It can be used to probe power and ground ports of an instance, even if the ports are defined as a global node, and do not appear in the subcircuit port list. The *`instance_port_name`* can be hierarchical, and can contain question marks and wildcards. For example: `x(x?1.*.n*.vdd)`
`vol = v(node1, node2)`	Probes the voltage difference between *`node1`* and *`node2`*, and assigns the result to the variable *vol*.

`expr` `=`
`par('`*`expression`*`')`
	Probes the expression of simple output variables and assigns the result to *`expr`*. The expression can contain variables in the above two formats, as well as all the mathematical operators, and built-in or user-defined functions. An expression can also contain the names of other expressions.

`depth =` *`value`*	Specifies the effective depth in the circuit hierarchy for the wildcard name. If it is set to `1`, only the nodes at the current level are applied (default value is infinity).
`subckt =` *`name`*	Specifies the subcircuit to which this statement applies. By default, it applies to the top level. If the statement is already in a subcircuit definition, this parameter is ignored. Setting this parameter is equivalent to defining the statement within a subcircuit declaration.
`exclude =` *`pn1`*, *`pn2`*	Specifies the output variables to be excluded from the probe. Names can be node or element names, and can contain wildcards.

.probe Option	Description
`preserve=none|all|port`	Defines the content of nodes probed with wildcard probing. `none` probes all nodes and ports connected to active devices (default). Nodes connected only to passive elements are not probed. `all` probes all nodes, including nodes connected to passive elements, and probes all ports. `port` only probes ports in subcircuits.
`domain=` `analog|digital|mixed`	Controls which signals (digital, analog, or both) would the `.probe` command select for probing. The default value is `analog`. For more information about the behavior of different values of the `domain` switch, see Description of domain Switch Values on page 90
`vcd=yes|no`	Enables you to choose `vcd` format for saving digital signals. This is especially useful if you do not use the Cadence waveform viewer and therefore cannot read the SST2 format. The default value is `no`. This option will have no effect when `domain` is set to `analog`. **Note:** The `vcd=yes` option can be used with `.probe` only for the UltraSim solver. The Spectre and APS solvers do not support this option.

Description of domain Switch Values

The following table describes the behavior of different values of the `domain` switch.

domain Value	Which Signals are Saved	Database and Format
`analog`	Only analog signals are probed and saved.	Analog signals are saved into the database format that is selected by the `rawfmt` option. If `rawfmt` is not specified, the default database format that is used is SST2 for AMSU and psfbin or SST2 for AMSS. The database to save the signals is selected from the analog control file (`[analog-control-file].raw` by default, or as specified by the `+amsrawdir` option).
`mixed`	Probes both analog and digital signals. If a signal belongs to a discrete domain, or to a scope that may contain further objects or scopes in discrete domains, this option will internally create a Tcl probe command for such objects or scopes with the `-domain` Tcl `probe` option set to `digital`. In case of digital signals, any function (such as `v` and `i`) applied on the signal is ignored.	For analog signals, database and format would follow the same behavior as described for the `domain=analog` setting. The digital signals are saved in SST2 format in the SHM database if the `vcd` option is not specified or if `vcd` is set to `no`. In case the SHM database is not already created, a default SHM database is created with the existing naming convention for default database. If the `vcd` option is set to `yes`, the digital signals are probed into a `vcd` database.

domain Value	Which Signals are Saved	Database and Format
`digital`	Only digital signals are saved. Signals are specified using either an explicit hierarchical name or a scope. If they are specified by scope, all digital objects under that scope would be considered for probing. Any function (such as `v` and `i`) applied on the signal is ignored.	For digital signals, database and format would follow the same behavior as described for the `domain=mixed` setting.

Description for .probe Statement

Sets up probes on nodes, ports, or elements for a specified output quantity. This statement can contain hierarchical names and wildcards for nodes, ports, or elements, and you can embed it within the scope of a subcircuit.

Note: When you use the AMS Designer environment, you will typically use Tcl commands instead of the `.probe` statement to set probes.

The following guidelines apply when you use the `.probe` statement and the UltraSim solver:

- The UltraSim solver uses capabilities, such as reduction, that effectively remove nodes from the design. You cannot probe such nodes and the solver issues a warning message if you try. Unfortunately, you cannot predict what nodes the software will remove.
- The UltraSim solver ignores nodes that are not part of the simulated circuit; any probes you create for these nodes will not have waveforms associated with them.
- The UltraSim solver creates a hierarchy of artificial component names down to the highest-level objects that it simulates. Error messages and wildcard probes sometimes return these artificial component names.
- The UltraSim solver creates artificial names for ports in the partitions of a design that are not simulated by the UltraSim solver. Error messages and wildcard probes sometimes return these artificial names.
- The UltraSim solver handles the names of all subcircuits, instances, and nodes referred to within a SPICE language module as lowercase names. This means that to probe such objects, you must use lowercase names even if the names in the SPICE module use uppercase or mixed-case names.

For example, you have a design that has a top-level Spectre module called `top` that instantiates a SPICE module instance called `spiceB1`, which, in turn, instantiates a Spectre module instance called `X2`. To probe into `X2`, you use a statement like

```
*ultrasim: .probe v(top.spiceB1.x2.A)
```

You specify the Spectre module instance using lowercase `x2` because this instance occurs in a SPICE module (`spiceB1`). You probe signal `A` using its original uppercase name because it occurs in a Spectre instance (`X2`), not in a SPICE instance.

- When probing current flow for a SPICE primitive instance,
  - The UltraSim solver probes the current for the first two ports only.
  - You cannot probe flow after starting the simulation, such as:

    ```
    tcl> run -sync
    tcl> probe -flow...
    ```

    Instead, you must create the flow probe prior to running the simulation.
  - The `probe` command generates different signal names when you use the UltraSim solver than when you use the Spectre solver (with the simulation front end parser). For example, consider the following design hierarchy:

    ```
                                 top
                                  |
                                 A1
                                  |
      +------------------------------------------------------------------+
      |         |         |           |             |           |        |
    resistor   myva   mybsim3v3   resistor_wrp   vares       iprobe   iprobe
      R2        X3        M3       inline_res     vares_inst   I2_1     I2_2
                |                     |             |
      +-------------------+     inline subckt    module vares(vp, vn);
      |         |         |
    iprobe    mysub     iprobe
    Isub_pos   Rsub      Iub_pos
                |
            spice subckt
    ```

    Here are the signal names for each solver:

AMS-UltraSim	AMS-Spectre (SFE)
`top.A1.X3.s1_$flow`	`top.A1.X3.s1_$flow`
`top.A1.X3.s2_$flow`	`top.A1.X3.s2_$flow`
`top.A1.a1_$flow`	
`top.A1.a2_$flow`	

AMS-UltraSim	AMS-Spectre (SFE)
`top.A1.M1.d_$flow`	`top.A1.M1.b`
`top.A1.M1.g_$flow`	`top.A1.M1.g`
(no probe)	`top.A1.M1.s`
`top.A1.M1.b_$flow`	`top.A1.M1.b`
`top.A1.R2.\1_$flow`	`top.A1.R2.\1`
`top.A1.R2.\2_$flow`	
`top.A1.I2_1.n_$flow`	`top.A1.I2_1.in`
`top.A1.I2_1.out_$flow`	`top.A1.I2_1.out`
`top.A1.I2_2.n_$flow`	`top.A1.I2_2.in`
`top.A1.I2_2.out_$flow`	`top.A1.I2_2.out`
`top.A1.X3.Rsub.1_$flow`	
`top.A1.X3.Rsub.pos`	
`top.A1.X3.Isub_neg.in_$flow`	`top.A1.X3.Isub_neg.in`
`top.A1.X3.Isub_neg.out_$flow`	`top.A1.X3.Isub_neg.out`
`top.A1.X3.Isub_pos.in_$flow`	`top.A1.X3.Isub_pos.in`
`top.A1.X3.Isub_pos.out_$flow`	`top.A1.X3.Isub_pos.out`
`top.A1.\X3:1`	
`top.A1.\X3:2`	
`top.A1.X3.Rsub.\neg:x0`	
`top.A1.inline_res.foo1.x0`	
`top.A1.inline_res.foo2.x0`	
`top.A1.inline_res.\1_$flow`	`top.A1.inline_res.\1`
`top.A1.inline_res.p`	
`top.A1.inline_res.\n:x0`	
`top.A1.vares_inst.vn_$flow`	`top.A1.vares_inst.vn`
`top.A1.vares_inst.vp_$flow`	`top.A1.vares_inst.vp`

Examples for .probe Statement

```
*ultrasim: .probe v(n1) i1(m1) vdiff = v(n2,n3) expr1 = par('v(n1)+2*v(n2)')
```

The above example, probes the voltage at node `n1` and the current `i1` for element `M1`. The voltage difference between nodes `n2` and `n3` is probed and assigned to `vdiff`. In addition, an expression of voltages at nodes `n1` and `n2` is probed and is assigned to `expr1`.

```
*ultrasim: .probe tran v(*) i(r1) depth = 2 subckt = VCO
```

The above example probes the voltages for all the nodes in the subcircuit named `VCO` and one level below in the circuit hierarchy. Also probes the current of the resistor `r1` for all the instances of the subcircuit VCO. The reported names of `r1` are appended with the circuit call path from the top level to VCO. This is equivalent to the situation where the statement

```
*ultrasim: .probe tran v(*) i(r1) depth = 2
```

is written in the subcircuit definition of `VCO` in the netlist.

```
*ultrasim: .probe tran  X(xtop1.block1.in) X0(xtop1.block1.in)
```

The above example reports currents for a subcircuit `block1`, which is instantiated in top-level block `xtop`. The `X` output variable returns the current into the subcircuit `in`, including all lower hierarchical subcircuit ports, while the `X0` output variable returns only the current into the subcircuit port and excludes all other lower hierarchical subcircuit contributions.

To probe the subcircuit instance port current, use the following format:

```
*ultrasim: .probe tran x0(xtop.x23.xinv.out)
```

The above example probes the current of port `out` of instance `xtop.x23.xinv`, excluding all other lower hierarchical subcircuit ports.

```
*ultrasim: .probe tran x(xtop.*)
```

The above example probes the current of ports for instance `xtop` and for all instances below it.

```
.probe v(top.i1.*) v(top.i2.x?1.*) domain=mixed
```

In the above example, the analog front-end that is processing the `.probe` statement will pass to AMS the following information:

- Scope name (`top` in this case)
- Relative hierarchical name from `[scope]` up to the point where wildcarding occurs (`i1` and `i2` in this case)

Based on this information, AMS will issue the following Tcl command:

```
probe -create -shm -database [database] -domain digital [scope].[hier-name] -depth
all
```

where `[database]` is the name of the database already created with the `database -create` command. If no database is created, the command will point to the default SHM database.

Note that the analog signals selected by the above wildcard rule will continue to be saved normally.

```
.probe v(top.i1.i2.d_in) domain=mixed vcd=yes
```

In the above example, if the `.probe` statement refers to a hierarchical name and if the name is not visible in the analog-related design, the analog front-end that is processing the `.probe` statement will send the name back to AMS. AMS will check if the name exists in the digital design database. If the name exists in the digital design database, it will create the following Tcl probe statement with the `-vcd` flag set.

```
probe -create -vcd -database [database] -domain digital [hier-name] -depth all
```

Note that if `domain` was not set in the original `.probe` statement or was set to `analog`, the analog simulator would ignore the `.probe` statement because the signal was not found in the analog circuit hierarchy.

As a more extended example, assume that you have a top-level Verilog-AMS module with the following contents.

```
module test(a0,a1,b0,b1,carryin,s0,s1,c2);
...
onebit X1 (a0_, b0_, carryin_, s0, c1);
onebit X2 (a1_, b1_, c1, s1, c2);
...
endmodule
```

The `onebit` object is a SPICE subcircuit:

```
.SUBCKT onebit A B carry-in out carry-out
X1 A B D nand
X2 A D 8 nand
...
.ENDS
```

The `onebit` object shares a file named `add2b.sp` with the `nand` object:

```
.SUBCKT nand A B Y
MM12 Y A vdd! vdd! PMOS W=16.8u L=0.6u M=1.0
MM11 Y B vdd! vdd! PMOS W=8.4u L=0.6u M=1.0
...
.ENDS
```

To probe values in the `nand` you use a command like this:

```
*ultrasim: .probe V(test.X1.x1.a)
```

The first `X1` in the probe retains its upper case letters because it occurs in the case-sensitive language of Verilog-AMS. However, the second `x1` must be all lower case because that is the name of an instance found within a SPICE object. Similarly, the signal name `a` becomes lower case because it too appears within a SPICE object.

Details about the .usim_opt Options the Software Adds Automatically

When you use the AMS Designer simulator with the UltraSim solver, the software automatically adds the following options (specifically for AMS Designer simulation) just prior to simulating:

```
.usim_opt del_allnode_inst=no
.usim_opt wf_output_format=verilog
.usim_opt wf_param_hier=1
.usim_opt addflowsuffix=yes
```

Do not change these options. These options have the following meanings:

.usim_opt	Description	
`addflowsuffix`	Specifies whether the software adds `_$flow` as a suffix to all current probe names. The AMS Designer simulator sets the value to `yes`. Valid Values:	
	`yes`	The software adds the `_$flow` suffix.
	`no`	The software does not add this suffix.
`del_allnode_inst`	Specifies whether the software removes (through UltraSim's reduction algorithm) elements having the same nodes connected to all terminals. The AMS Designer simulator sets the value to `no`. Valid Values:	
	`yes`	The software removes these elements.
	`no`	The software does not remove these elements.
	Note: For more information about UltraSim's reduction algorithm, see the *Virtuoso UltraSim Simulator User Guide*.	
`wf_output_format`	Specifies which naming convention the software uses to generate outputs. The AMS Designer simulator sets the value to `verilog`.	
	`spice`	The software generates outputs using SPICE naming conventions.
	`verilog`	The software generates outputs using Verilog naming conventions.

.usim_opt	Description	
`wf_param_hier`	Specifies how the software delimits *`name`*`:`*`terminal`* in current probe names. The AMS Designer simulator sets the value to `1`.	
	`0`	The software uses `:` to delimit the terminal part of the hierarchical name, and escapes last part of the hierarchical name (an instance). For example: `top.s1.\s2:1`
	`1`	The software uses `.` instead of `:` to delimit the terminal name as the last part of the hierarchical name, and escapes the last part of the hierarchical name (a terminal). For example: `top.s1.s2.\1`

The following options:

```
.usim_opt wf_param_hier=1
.usim_opt addflowsuffix=yes
```

causes the software to modify current probe names as follows:

```
top.s1.\s2:1
```

becomes:

```
top.s1.s2.\1_$flow
```

Mixed-Signal DC Initialization

The Virtuoso 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 and Ultrasim solvers 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
UltraSim	2

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
```

Note: True DC analysis is currently not available.

Time-Saving Techniques for the Analog Solvers

This section discusses different methods to reduce the time devoted to simulating the analog sections of a design. The topics discussed are

- Adjusting Speed and Accuracy on page 100
- Saving Time by Selecting a Continuation Method on page 100
- Specifying Efficient Starting Points on page 100

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 Virtuoso® 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 Virtuoso 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.

Note: 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 L1:1=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`

Supplying Nodesets

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 L1:1=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 1µ.

Specifying 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 `ic` statements are used, and the `ic` parameter settings on the capacitors and inductors are ignored.
`dev`	The `ic` parameter settings on the capacitors and inductors are used, and the `ic` statements are ignored.
`all`	Both the `ic` statements and the `ic` parameters are used. If specifications conflict, `ic` parameters override `ic` 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.

5

Using an amsd Block

An `amsd` block contains statements that control AMS mechanisms during the elaboration phase of `irun` (or during `ncelab`) 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 `irun` 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.

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

Tip

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 AMSCB 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. See also "Migrating to an amsd Block from prop.cfg" on page 639.

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
}
```

Important

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

Note: 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
portmap	The `portmap` statement tells the AMS Designer simulator how a SPICE subcircuit interface should appear to the elaborator
config	The `config` statement specifies which definition you want to use for particular cells or instances.
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.
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.
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:

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
}
```

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]
```

Important

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 to which to apply these `portmap` settings. Use this form (typically, together with `config use=spice`) when your design contains SPICE-on-leaf constructs. Valid Values: Any valid subcircuit name. **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 `portmap` settings. Use this specifier (typically, together with `config use=hdl`) 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 `portmap` settings. Use this specifier (typically, together with `config use=hdl`) 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: `yes`

	See also “Binding Ports using autobus” on page 314.
`excludebus`	List of entities not to map because they are not buses. No default.
`reversebus`	Specifies the `reversebus` name. No default.
`busdelim`	List of bus delimiters. Valid Values: `[]`, _, `<>`, `none`, or any single character Default: `[] <>`
`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: `lower`

`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: `keep`

`file`	Name of the port-bind file containing customized port bindings. See also “Binding Ports using a Port-Bind File” on page 316.

`reflib`	Name of the library of precompiled cell containing custom port bindings. If the `reflib` parameter is provided with the `portmap` statement, the `reffile` parameter is not compiled even if it is specified with the `portmap` statement. The library of precompiled cell provided through the `reflib` parameter will be used instead of the `worklib` library.

The use of `reflib` 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 `reflib` parameter works only for VHDL-SPICE, and not for Verilog-SPICE.

In the example given below, the software will search for the precompiled cell `sl1` in the library `mylib`. Notice that the `reffile` parameter is not specified with the `portmap` statement.

```
portmap subckt=sl1 refformat=vhdl reflib=mylib
config cell=sl1 use=spice
```

`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 `portmap ... reffile=/user/project/myfile.v ...`

```
portmap ... reffile=/user/project/myfile.v ...
```

- Use `refformat` to specify the format of this file. (The default format is `verilog`.)
- Use `porttype` 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 `reffile` parameter with the `portmap` card if you have specified it with the `irun` command, unless you want to use a different `reffile` than what you specified to `irun`.

`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:

`name`	Match ports by name.
`order`	Match ports by order.

Default: `order`

`refformat`	Specifies the format of the `reffile`. Valid Values: `verilog` — Verilog format See also "Binding Ports using a Verilog File" on page 316. `vhdl` — VHDL format Default: `verilog`
`stub`	Name of the cell for which you want to use the stub version; you must also specify a `match` assignment.
`match`	Specifies which definition to use when matching the interface. The `match` assignment applies only when you also specify `stub` on the `portmap` statement and `use`=`stub` on the `config` statement. Valid Values: `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:

```
amsd{
      ie vsup=1
      portmap subckt=invx2 porttype=name ignore=vdd vss
      config cell=invx2 use=spice
    }
```

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 will suffice. The tool will automatically generate implicit portmaps, with default values, for the subcircuits or modules referred to in the `config` card. In the autogenerated portmap statement, the cell name used in the `config` statement will be used as the SPICE subcircuit name or the Verilog module name, and the default values will be assumed for all other parameters of the `portmap` statement.

For example, if the `config` statement is:

```
config cell=divider use=hdl
```

The autogenerated `portmap` statement will be:

```
portmap module=divider
```

Similarly, if the `config` statement is:

```
config cell=analog_top_a use=spice
```

The autogenerated `portmap` statement will be:

```
portmap subckt=analog_top_a
```

The tool will assume the following `portmap` parameter default values for autogenerating skeletons, commfile, or the portbind file:

```
autobus=yes
busdelim="[] <>"
refformat=verilog
interconnect=mixed
```

Note: If you want to use a non-default value for any of the above parameters or if you want to use `reffile` or `file` parameter with the `portmap` card, you need to use 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 (shown in the example below).

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

  where `inst1`, `inst2`, and `inst3` are instances of `foo` in the Verilog module `top`, which need to be configured to 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 (see SPICE Port to Verilog Bus Mapping Example on page 116):

- Using the reference file name with the `irun` 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_ULOGIC
                                             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=anglog_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=anglog_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=pll_top busdelim=_ file=pll_top.pb
    config cell=pll_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 pll_top(refclk, p0_clk, reset);
wire [1:0] p0_clk;
...
endmodule


module top;
...
pll_top p1(...);
...
```

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`:

1. Using the reference file name with the `irun` command:

   ```
   irun 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 `irun` command. The port `pll_clk` in the resulting port bind file will be mapped as:

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

2. 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 `pll_clk` in the resulting port bind file will be mapped as:

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

3. 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 `pll_clk` in the resulting port bind file will be mapped as:

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

4. 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 `pll_clk` in the resulting port bind file will be mapped as:

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

5. Using `file` in portmap card

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

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

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

config

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

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

Important

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 (`use=`*`definition`*). If you specify more than one cell name, you must use double quotation marks around the list:
	`config cell="subA subB subC" use=spice`
	`config cell=dsp use=hdl`
	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.
	`config inst="top.inst1 top.inst2 top.inst3" use=spice`
	In addition, if the hierarchical path of an instance contains an index (vector bit), you must use double quotation marks around the instance definition.
	`config inst=”top.inv_chain.forgentblk[0].case_blk0.inv” use=spice`

`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 `cell` parameter assignment; you can specify a `match` choice of `verilog` or `spice` in the `portmap` statement for the stub

No default. You must specify a `use` assignment.

`exclude`

Specifies a scope that will be excluded from the application of the `use` parameter setting in the `config` card.

In the example below, the `use=hdl` setting will be applied to all instances of cell `inv` except those within the scope `mid2_block`, which is a block within `mid_block`.

```
amsd {
...
config cell=mid_block use=spice
config cell=inv use=hdl exclude=mid2_block
}
```

Important

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
    }
```

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.

Note: 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]
```

It is mandatory to specify either `vsup=`*`supplyValue`* or `connrules=inhconn_full` in the `ie` statement. `vsup=`*`supplyValue`* specifies the final real number for logical 1, whereas `connrules=inhconn_full` uses inherited connection properties for power supply and ground nets for connect modules.

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`

These built-in connect rules are regular connect rules with certain design style so that the connect rule parameters are all 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 customizations 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 `irun.log` file.

See

- Scope Assignments on page 121
- Parameter Assignments on page 123
- Examples on page 126

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.
`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: `ie vsup=1.2 instport="a b c"`
`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.
`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.

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.

Note: 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`".

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:

`vthi`	Voltage value above which the simulator assigns a logical `1`. The simulator determines the default value from the connect rule.
`vtlo`	Voltage value below which the simulator assigns a logical `0`. The simulator determines the default value from the connect rule.
`vx`	Final real number for logical `x`. The simulator determines the default value from the connect rule.
`tr`	Rise time for analog transition, from `vtlo` to `vthi` or `vx`. Default Value: 0.2 ns
`rlo`	Output resistance for L2E when digital input is `0`. Default Value: 200 Ohms
`rhi`	Output resistance for L2E when digital input is `1`. Default Value: 200 Ohms
`rx`	Output resistance for L2E when digital input is `x`. Default Value: 40 Ohms
`rz`	Output resistance for L2E when digital input is `z`. Default Value: 10M Ohms

`txdel`	Controls the amount of wait time before a digital port is driven to `x` for the connect modules `E2L`, `E2L_2`, `L2E`, `Bidir`, and `Bidir_2`.
	Measured in nano seconds.
	Default Value: Four times the `tr` parameter. If the `tr` parameter is not specified for the `ie` statement, the simulator uses the default `tr` value 0.2n and calculates the default `txdel` value as 0.8n.
	Example:
	`amsd {`
	`        ie vsup=1.8 txdel=0.9n`
	`}`

`mode`	Specifies the type of connect module insertion in the `ie` card. You can assign two values to this parameter: ■ `merged` - 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. ■ `split` - The `split` 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.
`vpso`	Converts the PSO X state to a user-supplied voltage in AMS-CPF.
`vdelta`	Voltage delta value ranging from 0 to vsup. Default Value: vsup/64
`vtol`	Voltage tolerance value ranging from 0 to vdelta. Default Value: vdelta/4

`connrules`	Connect rule to build using the `vsup=`*`supplyValue`* 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 `inhconn_full` argument can inherit supply voltage automatically. The default supply is `cds_globals.\vdd!` and `cds_globals.\vss!` You can use this argument instead of using the `vsup` parameter to specify the supply voltage. However, if both `vsup` and `connrules=inhconn_full` exist in the same `ie` card, an error is reported. For example: `ie vsup=1.8 connrules = inhconn_full`
`full_fast`	Build the "full-fast" connect rule

Default Value: `full_fast`

Note: You can find the set of connect rule files that Cadence provides—including the "`basic`", "`full`", "`inhconnfull`", and "`full-fast`" connect rules—in *`your_install_dir`*`/tools/affirma_ams/etc/connect_lib`. See *`your_install_dir`*`/tools/affirma_ams/etc/connect_lib/README` for detailed information about them.

`discipline`

Used as the discipline name corresponding to the `ie` card. If this parameter is not specified, the discipline name is auto-generated.

In the example below, the discipline name corresponding to the `ie` card in the first `ie` statement will be `logic1_8` instead of the auto-generated discipline name `ddiscrete_1_8`. However, because the `discipline` parameter is not used with the second and third `ie` statements, the corresponding discipline names for these `ie` cards will be `ddiscrete_3_3` and `ddiscrete_4_5`.

```
amsd {
    ie vsup=1.8 discipline=logic1_8 rx=25
    ie vsup=3.3 rlo=125
    ie vsup=4.5 tr=0.4n
    }
```

Important

The `discipline` parameter can take only discrete discipline values. In the example below, the discipline value is not discrete and therefore it is not supported.

```
ie vsup=1.8 instport="top.sub.pin"
discipline="electrical"
```

For continuos disciplines, use the elaboration `-setdiscipline` option instead. For example:

```
-setdiscipline "instterm-top.sub.pin- electrical"
```

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

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

}
```

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:

`library.cell`
`library.cell``:arch`

For example:

```
name=mylib.std_logic2e
name=mylib.std_logic2e:arch
```

`type` — Digital signal type that connects to SPICE; for example:

```
type=std_logic
```

`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 `ce` statement; for example:

```
cell=dummy_spice
```

`genericmap`

Parameter for overriding generic values in the conversion element; for example:

```
genericmap="vsup 3.3 vthi 2.2"
```

`cellport`

(Optional) Name of the port (or ports) to which the elaborator will apply the parameter settings specified in the `ce` statement; for example:

```
cellport="vdd2 gnd2"
```

`excludeport`

(Optional) Name of the port (or ports) that will not use the parameter settings specified in the `ce` statement; for example:

```
excludeport="p0 nx"
```

`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:

```
ce ... type=base  priority=low
ce ... type=derived  priority=high
```

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 `high` priority.

`optimize`

(Optional) The global switch to turn off conversion element optimization. Default is `on`.

The following statement for the global `ce` card implies that conversion element optimization will be ignored for the entire design.

```
ce optimize=off
```

The following statement, on the other hand, implies that optimization will be ignored for scope `nand2` and below only. Conversion element optimization will occur for all other scopes.

```
ce optimize=off cell=nand2 ...
```

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 on page 248
  - fourstate on page 249
  - sum on page 251
  - avg on page 252
  - min on page 253
  - max on page 254
  - `none`

Note: 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=logic1_8 rx=25
    ie vsup=3.3 rlo=125
    ie vsup=4.5 tr=0.4n
    connectmap resolution=max discipline="logic1_8 ddiscrete_4_5"
    }
```

The first `ie` statement in the above example explicitly associates the `logic1_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 `logic1_8` discipline and the auto-generated `ddiscrete_4_5` discipline.

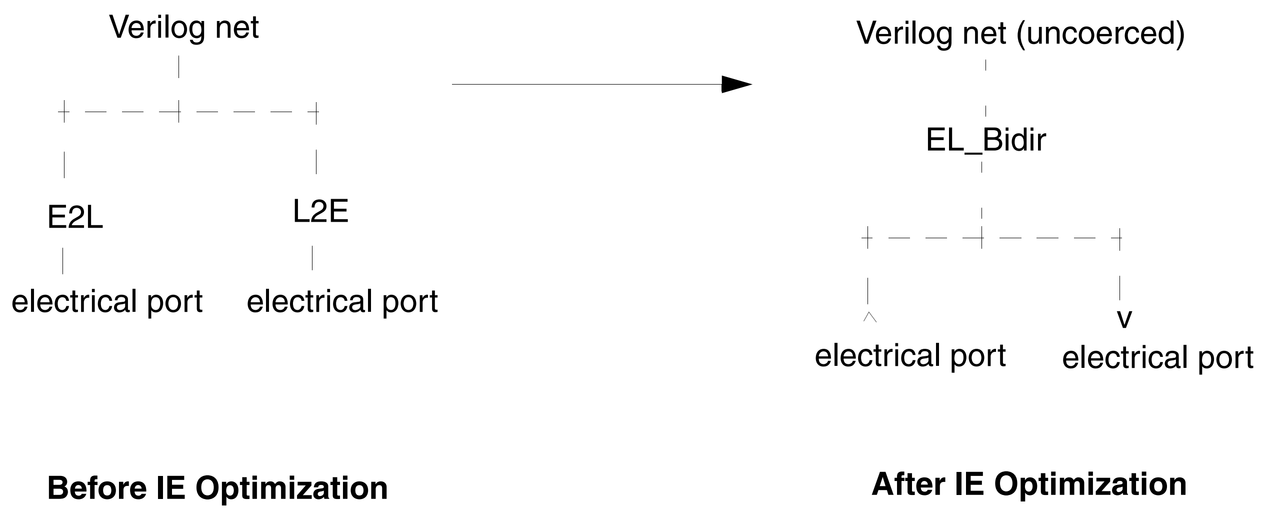
In addition to the explicitly defined resolution functions, the global and default resolution functions apply to `ie` custom and auto-generated disciplines as well.

Hierarchical Interface Element Optimization

Single-Level IE Optimization

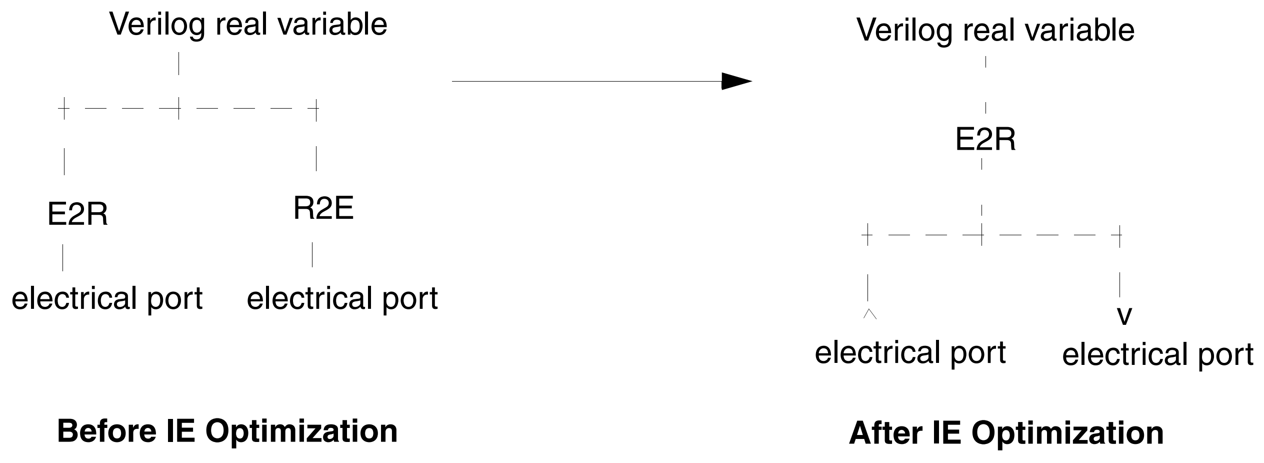
Single-level IE Optimization is a process in which the elaborator inserts a single bidirectional interface element (IE) in place of multiple IEs when the following conditions are met:

- 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.



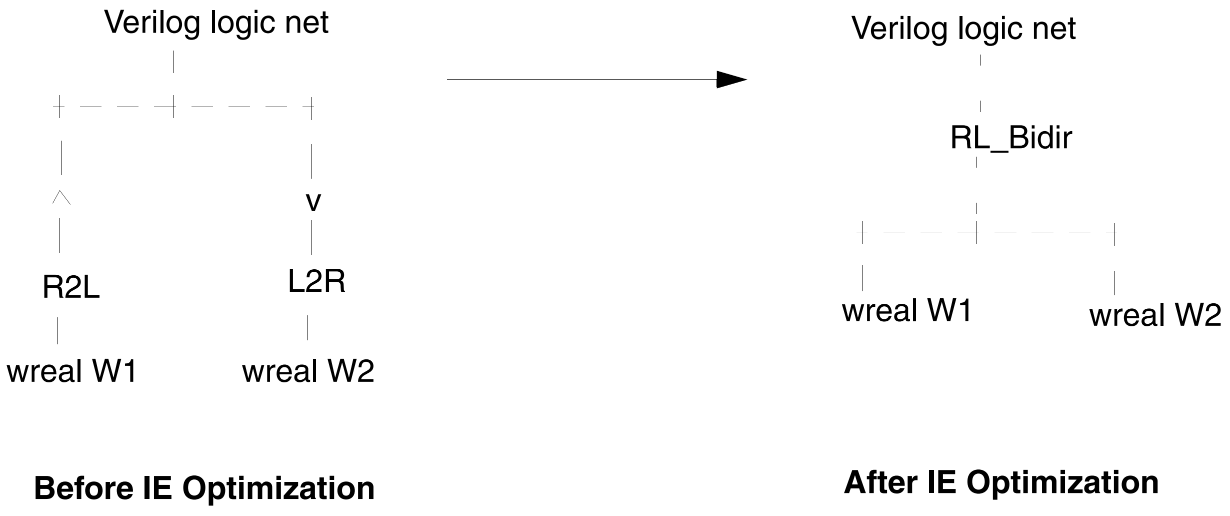
In case of analog/digital variable connections, multiple dissimilar IEs are replaced with a single electrical-to-real (E2R) IE that is connected to all the analog ports when:

- 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

```
            digital                              digital
            /     \                              /     \
           /       \                            /       \
          /         \                          /         \
         /           \                        /           \
+--------------+ +--------------+  +--------------+ +--------------+
|       |      | |      |       |  |       |      | |      |       |
|    digital   | |   digital    |  |    digital   | |   digital    |
|       |      | |      |       |  |       |      | |              |
|      E2L     | |     L2E      |  |     Bidir    | |              |
|       ^      | |      |       |  |       |      | |              |
|       |      | |      v       |  |       |------|-|------+       |
| +----------+ | | +----------+ |  |       |      | |      |       |
| |  analog  | | | |  analog  | |  | +----------+ | | +----------+ |
| +----------+ | | +----------+ |  | |  analog  | | | |  analog  | |
+--------------+ +--------------+  | +----------+ | | +----------+ |
                                   +--------------+ +--------------+
```

Before Hierarchical Optimization **After Hierarchical Optimization**

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

Digital Sandwich

```
   Analog A
       |
       v
      E2L
       |
       v
+--------------+
|   Digital    |
|      |       |
|      v       |
|     L2E      |
|      |       |
|      v       |
|  +--------+  |
|  |Analog B|  |
|  +--------+  |
+--------------+
```

Before Hierarchical Optimization

```
    Analog A--------+
                    |
                    |
+--------------+    |
|   Digital    |    |
|      |       |    |
|      v       |    |
|    Bidir     |    |
|      |       |    |
|      +-------|----+
|      |       |
|      v       |
|  +--------+  |
|  |Analog B|  |
|  +--------+  |
+--------------+
```

After Hierarchical Optimization

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

Logic Over Wreal

```
             logic                              logic
            /     \                            /      \
           /       \                          /        \
          /         \                        /          \
         /           \                      /            \
+--------------+ +--------------+  +--------------+ +--------------+
|       |      | |       |      |  |       |      | |      |       |
|     logic    | |     logic    |  |     logic    | |    logic     |
|       |      | |       |      |  |       |      | |              |
|      R2L     | |      L2R     |  |     Bidir    | |              |
|       ^      | |       |      |  |       |      | |              |
|       |      | |       V      |  |       |------|-|------+       |
| +----------+ | | +----------+ |  |       |      | |      |       |
| |   wreal  | | | |   wreal  | |  | +----------+ | | +----------+ |
| +----------+ | | +----------+ |  | |   wreal  | | | |   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

```
    wreal A
       |
       V
      R2L
       |
       V
+--------------+
|    Logic     |
|      |       |
|      V       |
|     L2R      |
|      |       |
|      V       |
|  +---------+ |
|  | wreal B|  |
|  +---------+ |
+--------------+
```

Before Hierarchical Optimization

```
         wreal A--------+
                        |
                        |
+--------------+        |
|    Logic     |        |
|      |       |        |
|      V       |        |
|    Bidir     |        |
|      |       |        |
|      +-------|---+----+
|      |       |
|      V       |
|  +---------+ |
|  |wreal B |  |
|  +---------+ |
+--------------+
```

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

```
              logic                                logic
            /       \                            /       \
           /         \                          /         \
          /           \                        /           \
         /             \                      /             \
+--------------+ +--------------+    +--------------+ +--------------+
|       |      | |      |       |    |       |      | |      |       |
|     logic    | |    logic     |    |     logic    | |    logic     |
|       |      | |      |       |    |       |      | |              |
|      R2L     | |     L2R      |    |     Bidir    | |              |
|       ^      | |      |       |    |       |      | |              |
|       |      | |      V       |    |       |------|-|------+       |
| +----------+ | | +----------+ |    |       |      | |      |       |
| | real var | | | | real var | |    | +----------+ | | +----------+ |
| +----------+ | | +----------+ |    | | real var | | | | real var | |
+--------------+ +--------------+    | +----------+ | | +----------+ |
                                     +--------------+ +--------------+
```

Before Hierarchical Optimization **After Hierarchical Optimization**

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

Wreal Over Electrical

```
              wreal
             /     \
            /       \
           /         \
          /           \
+--------------+ +--------------+
|       |      | |       |      |
|     wreal    | |     wreal    |
|       |      | |       |      |
|      E2R     | |      R2E     |
|       ^      | |       |      |
|       |      | |       v      |
| +----------+ | | +----------+ |
| |  analog  | | | |  analog  | |
| +----------+ | | +----------+ |
+--------------+ +--------------+
```

Before Hierarchical Optimization

```
              wreal
             /     \
            /       \
           /         \
          /           \
+--------------+ +--------------+
|       |      | |       |      |
|     wreal    | |     wreal    |
|       |      | |              |
|     Bidir    | |              |
|       |      | |              |
|       |------|-|------+       |
|       |      | |      |       |
| +----------+ | | +----------+ |
| |  analog  | | | |  analog  | |
| +----------+ | | +----------+ |
+--------------+ +--------------+
```

After Hierarchical Optimization

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

Wreal Electrical Sandwich

```
   Analog A
       |
       V
      E2R
       |
       V
+--------------+
|    wreal     |
|      |       |
|      V       |
|     R2E      |
|      |       |
|      V       |
|  +--------+  |
|  |Analog B|  |
|  +--------+  |
+--------------+
```

Before Hierarchical Optimization

```
  Analog A--------+
                  |
                  |
+--------------+  |
|    wreal     |  |
|      |       |  |
|      V       |  |
|    Bidir     |  |
|      |       |  |
|      +-------|--+
|      |       |
|      V       |
|  +--------+  |
|  |Analog B|  |
|  +--------+  |
+--------------+
```

After Hierarchical Optimization

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

Hierarchical IE optimization is enabled using the `-amsoptie` option on the irun command line, as follows:

```
irun -amsoptie
```

The following are the limitations of Hierarchical IE optimization:

- IE optimization does not occur for cases of a real variable or an electrical net over multiple logic ports.
- Logic nets connected to bidirectional ports alone are not sufficient to enable IE optimization. Unidirectional ports (both input and output ports) must be present for IE optimization to work.
- Hierarchical optimization does not occur for cases of an analog net over a digital net and for cases of a real variable over an electrical net.

6

Preparing the Design: Using Analog Primitives and Subcircuits

See the following topics for information about how you can instantiate analog primitives and subcircuits in your design.

See also “Compiled C Flow” on page 151.

Instantiating Analog Primitives

The supported analog primitives for the Virtuoso 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.

For more information about the primitives, see the “Components Statement” chapter of *Virtuoso Spectre Circuit Simulator Reference*.

You can instantiate these analog primitives with *ports* that you bind either by name or by order. See “Binding Ports by Name” on page 144 and “Binding Ports by Order” on page 145 for more information. See also “Guidelines for Binding Ports by Name or Order” on page 146.

Note: You cannot mix these two types of port bindings for a particular analog primitive instantiation.

You must always bind the *parameters* of an instantiated analog primitive by name. See “Binding Parameters” on page 146.

Binding Ports by Name

You can connect an instance of an analog primitive by explicitly 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 *Virtuoso Spectre Circuit Simulator Reference*.

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
endmodule

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);
```

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
```

Binding Parameters

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	`// 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=spline`
	`relerr=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));`

Automatically-sized parameter arrays are passed differently	`// Spectre`
	`Filter in gnd out gnd`
	`svcvs poles=[1 0 5 0]`
	
	`// Verilog-AMS`
	`svcvs #(.poles({1,0,5,0}))`
	`Filter(in,ground,out,ground);`

Instantiating Analog Primitives (UltraSim Solver Only)

You can instantiate analog primitives in your structural Verilog-AMS modules. In this context, an analog primitive is defined as a SPICE or Spectre subcircuit, or as a Verilog-A component brought into the design by using an `ahdl_include`.

The UltraSim solver supports and recognizes all Spectre built-in primitives except for the following ones:

Unsupported primitive	Reason the primitive is unsupported
`a2d`	For use only by verimix-mixsim application.
`d2a`	For use only by verimix-mixsim application.
`node`	Infinite number of ports.
`prst`	Infinite number of ports.

The terminals of an analog component are always evaluated as `inout`.

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.

Note: 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.

Specifying 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-of-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
```

You require the `-ams_generate` command-line option to enable the Verilog-AMS generate flow.

Currently, the following is not supported:

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

Including 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` file

  For more information, see "Using hdl.var Variables with ncelab" on page 436.

- Use the `-modelpath` option for `ncelab`

  For more information, see "-modelpath Option" on page 423.

To include a subcircuit in your design, do the following:

➤ Use the `prop.cfg` file to reference subcircuit files.

Note: Do not use the `prop.cfg` file to reference model libraries or 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!
```

Compiled C Flow

The compiled C flow is turned on by default in AMS for both Spectre and UltraSim solvers.

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 stores the compiled code and the shared objects in a new directory that gets created in the directory where the simulation runs. For more information, see "Using the Compiled C Code Flow" in the *Cadence Verilog-A Language Reference*.

You can turn off the compiled C flow by doing the following (assuming a C-shell):

- At the shell command line, type

```
setenv CDS_AHDLCMI_ENABLE NO
```

To turn the compiled C flow back on, do the following (assuming a C-shell):

- At the shell command line, type

```
setenv CDS_AHDLCMI_ENABLE YES
```

The AMS Designer simulator does not provide an option equivalent to `spectre -ahdlcom:` You can enable or disable the compiled C flow only by setting the `CDS_AHDLCMI_ENABLE` environment variable.

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.

Note: The simulator recompiles the code under the following condition:

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

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.

For more information, see *Specifying Multi-Threading Options* in the *Virtuoso Spectre Circuit Simulator* and *Virtuoso Accelerated Parallel Simulator* User Guides.

Using the mtline Component with the AMS Simulator

AMS Designer supports using multiconduction transmission line (mtline) components in your designs. (For information about the mtline component, see the *Virtuoso Spectre Circuit Simulator Reference*.) See the following topics for information about how to use the mtline component with the AMS Designer simulator:

- Using mtline in a Schematic on page 153
- Using mtline in a SPICE or Spectre Subcircuit or Model on page 154

Using mtline in a Schematic

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.

   For guidance, see “Modeling Transmission Lines Using the LMG GUI,” in chapter 7 of *Spectre RF User Guide*.
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.

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.

7
Preparing the Design: Using Mixed Languages

When preparing your design using mixed languages, you should consider the following topics:

- Fetching Values Associated with an Analog Object on page 225
- Using the Strength-Based Interface Element (SIE) on page 228

See also “Using the wreal Data Type” on page 232.

Importing Verilog-AMS Modules into VHDL

Important

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, you must first 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. To generate these shells, you use the `ncshell` command.

Note: 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:

```
ncelab: *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.
```

See the following topics for details:

- Using the ncshell Command on page 158
- Performing the Steps to Import a Verilog-AMS Module on page 160

Using the ncshell 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 `ncshell` command, see "Generating a Shell with ncshell" in the “Mixed Verilog/VHDL Simulation” chapter of *Cadence VHDL Simulation User Guide*.

`ncshell -import verilog -ams -into vhdl` `[`*`other_options`*`]` *`lib.cell`***`:`***`view`*

`-import verilog`	Indicates that the imported model is Verilog, Verilog-A, or Verilog-AMS
`-ams`	Indicates that the imported model is a Verilog-AMS (or Verilog-A) module
`-into vhdl`	Indicates that you want to import the model into a VHDL module
`other_options`	Zero or more of the following options or the options described in “ncshell Command Options” in the “Mixed Verilog/VHDL Simulation” chapter of *Cadence VHDL Simulation User Guide*

`-package` *`name`*	Allows you to specify the package name for the VHDL package that `ncshell` generates.
`-view` *`name`*	Allows you to specify the cellview name to use for the digital Verilog shell and for the architecture name of the VHDL shell.
`-generic`	If the imported Verilog-AMS module has parameters, you must use this option to make those parameters available in the shell.

`lib.cell``:`*`view`*	Specifies the compiled design unit you want to import **Note:** If you have not compiled the design unit, you can add the `-analyze` option to the `ncshell` 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:

```
ncshell -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:

```
ncshell -import verilog -ams -into vhdl -generic -analyze comparator.vams
    comparator
```

Important

`ncshell` does not support the Verilog-AMS `wreal` port type.

Performing the Steps to Import a Verilog-AMS Module

To import a Verilog-AMS module with explicitly declared analog ports into VHDL, do the following:

1. Use the `ncvlog` compiler to compile the Verilog-AMS source code for the module that you want to import.
2. Use the `ncshell` 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 `ncshell` 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 `ncshell` 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 `ncvhdl`.
6. Elaborate the design using the `ncelab` elaborator.

For example, to import the Verilog-AMS module in the file `comparator_analog.v`:

```
‘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
```

do the following:

1. Compile the Verilog-AMS source code.

   ```
   ncvlog -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 `ncshell`. The argument to the `ncshell` command is the Library.Cell:View specification for the compiled module.

```
ncshell -import verilog -ams -into vhdl -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 `ncshell` utility also generates the following VHDL component declaration in the file `comparator_comp.vhd`:

```
library ieee;
use ieee.std_logic_1164.all;

package HDLModels is

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 HDLModels;
```

Note: 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 `ncshell`.

```
ncvhdl -v93 comparator_comp.vhd top.vhd -use5x
```

5. Elaborate the design with `ncelab`. 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

```
ncelab amslib.top:config AMSconnect -discipline logic
```

In this example, using the `-discipline logic` option sets the discipline of the shell ports to logic.

Connecting 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.

  Note: 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 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 trangate 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

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. For more information, see "Mapn2d" on page 350.

Using 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.

Note: 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
```

Using 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.

Connecting VHDL Blocks to SPICE Blocks

Direct instantiation of SPICE blocks within a VHDL scope is supported only using the `irun` 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` statements 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 multiplicant 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 IS
END  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"
}
```

See "ce" on page 129 for more information.

Instantiating 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 on page 170
- Component Instantiation of SPICE Primitives in VHDL-Digital on page 171
- Direct Instantiation of SPICE Primitives in VHDL-AMS on page 172
- Instantiation of SPICE Primitives Through SPICE Models on page 172

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.

Note: 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 `irun` command, as shown in the following example.

```
irun -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);
```

Note: You can also specify model files for an analog device using the `include` statement in the AMS control file.

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.

Obtaining CE Information in VHDL-SPICE

In AMS Designer simulator, you can use the elaborator option called `-cereport` to generate a report with information about the final set of CEs inserted in a VHDL-SPICE design. Note that 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

Note: 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

Note: 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.

Using Verilog-AMS Connect Modules for VHDL-SPICE Connection

You can use the AMSD 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.

Note: 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.

```
irun -compile `cds_root irun`/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 ncelab or irun option `-use_cm`.

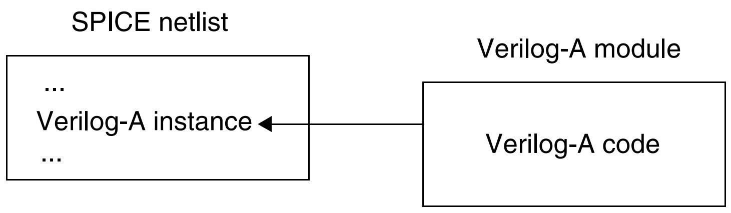
Currently, this feature supports only `std_logic` and electrical connections.

The following are not supported:

- VHDL real and SPICE electrical connection
- VHDL subtypes of std_logic and VHDL record types

Using 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 or with the UltraSim solver, 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 Verilog-A language models.

Important

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`.

Using 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 blocks.

To run a SPICE-on-Top design, you use the `-spicetop` option with the `irun` command.

For example:

```
irun xtop.scs foo.v <other irun options> -spicetop
```

where `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
     }
```

Using 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.

Using Spice Blocks in the Middle of Verilog-AMS 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 or with the UltraSim solver, you can specify the Verilog-AMS module that you instantiate in the SPICE block with 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
    }
```

Note: 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.

Important

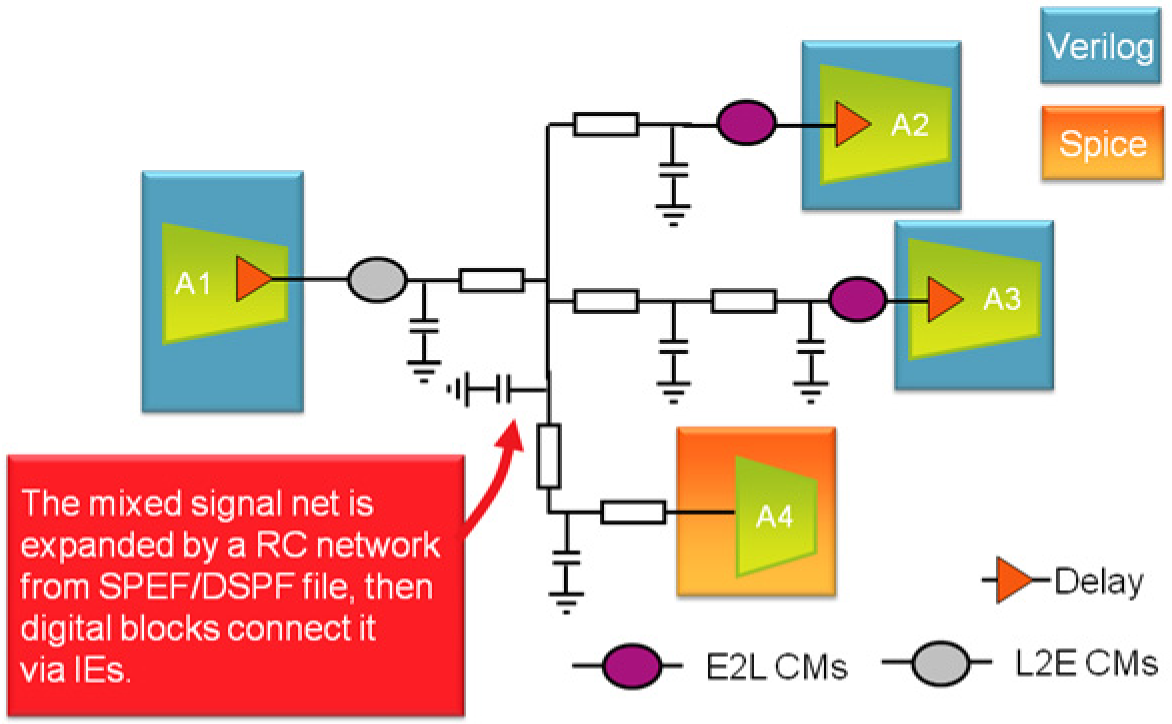
If you use the `reffile` parameter on a `portmap` statement to specify custom port bindings for a SPICE-in-the-middle arrangement, the reference file must satisfy the following requirements:

- ❑ 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`).

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 (CMs), as shown in Figure 7-1 on page 179.

Figure 7-1 RCs from SPEF are Expanded on Mixed-Signal Nets


Verilog
Spice
A1
A2
A3
A4
The mixed signal net is expanded by a RC network from SPEF/DSPF file, then digital blocks connect it via IEs.
Delay
E2L CMs
L2E CMs

To correctly specify SPEF or DSPF stitching on a mixed-signal net in AMS-APS and AMS-UltraSim, 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
}
```

Note: 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 `ncelab` optimizes this node to logic discipline even if it lies in the SPICE block.

4. Set up the `-amsspef` option in the `irun` arguments to control this feature, as shown below.

```
irun   ./*.v \
        ./config.scs \
        ./cds_globals.vams \
        ./top.vams \
        -amsfastspice \
        -input ./probe.tcl \
        -sdf_cmd_file ./command_sdf \
        -amsspef
```

Note: This feature is supported in AMS-UltraSim, AMS-APS, and AMS-Spectre solvers.

Note: The behavior of AMS-APS stitching is similar to standalone APS stitching. For example, the `*.spfrpt` file is dumped into the simulation directory.

Using 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.

Note: 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;
```

Connecting Verilog-AMS Vector Buses to 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" on page 374, "sourcefile_opts Property" on page 375, and "The Port Mapping File" on page 383.

Using Port Expressions when Connecting to Analog

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 );
```

Accessing SPICE Nets inside a Verilog Design

You can use the following special instances to access SPICE nets in a Verilog design:

Instance Type	Description
`cds_spice_a2d`	Analog-to-digital connection from SPICE to Verilog (analog drives)
`cds_spice_d2a`	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 testbench

dut 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:

```
module dut(in1, in2);          // Design Structure
blockA A(in1);                 //
blockB B(in2);                 //            TB (the testbench)
endmodule                      //             |
                               //            DUT
module blockA(in);             //             |
blockC CC(in);                 //      ---------------
blockD DD(in);                 //      |             |
endmodule                      //      A             B
                               //      |             |
module blockB(in);             //   ------           EE
blockE EE(in);                 //   |    |
endmodule                      //   CC   DD

module blockCC(in);
endmodule

module blockDD(in);
reg a,b,c,d;
reg f;
...etc
endmodule

module blockEE(in);
endmodule
```

Perhaps the SPICE representation of `blockA` looks something like this:

```
subckt blockA(in)             // Design Structure with blockA as SPICE
X1 in blockCC                 //
X2 in blockDD                 //             TB (the testbench)
ends                          //              |
                              //             DUT
subckt blockDD(in)            //              |
X1 in a                       //       ---------------
X2 a b   blockXXX             //       |             |
X3 b c   blockYYY             //       A             B
X4 c d f blockZZZ             //       |             |
ends                          //    ------           |
                              //    |    |           EE
                              //    X1   X2
```

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`	`module cds_spice_a2d(e);`
	`output e;`
	`electrical e;`
	`parameter spicenet = "null";`
	`endmodule`
		analog-to-digital

`cds_spice_d2a`	`module cds_spice_d2a(e);`
	`input e;`
	`electrical e;`
	`parameter spicenet = "null";`
	`endmodule`
		digital-to-analog

Instance Type	Effective Module Definition	Interface Element

`cds_spice_bidir`	`module cds_spice_bidir(e);`
	`inout e;`
	`electrical e;`
	`parameter spicenet = "null";`
	`endmodule`
		bidirectional

`cds_spice_a2a`	`module cds_spice_a2a(e);`
	`input e;`
	`electrical e;`
	`parameter spicenet = "null";`
	`endmodule`
		none

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 a Command-Line Option to Manage Out-of-Module References to SPICE on page 188
- Using Compiler Directives to Manage Out-of-Module References to SPICE on page 189

Using a Command-Line Option to Manage Out-of-Module References to SPICE

You can use the `-ignore_spice_oomr` and `-default_spice_oomr` command-line options 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. 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)

When you use the `-ignore_spice_oomr` command-line option, the software ignores any digital statements that contain out-of-module references to SPICE blocks.

In the following example, when you use the `-ignore_spice_oomr` command-line option, the software ignores `$display("1")` and `$display("2")`, but `$display("3")` remains active:

```
if ( spice_oomr_ref )
     $display("1");
else
     $display("2");
$display("3");
```

When you use the `-default_spice_oomr` command-line option, the software assigns a default value (`1'bx`) 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 `-default_spice_oomr` command-line option causes the software to set the value in the statement to `1'bx`.

```
module testbench ();
  ...
  wire vcoclk, clock_2, clock_1, clock_0, net036, p0;
  ...

  initial begin
    $monitor (testbench.p1.p0);
  end

  ...
  pll_top p1(refclk, reset, vcoclk, clock_2, clock_1, clock_0, net036, p0,
             clk_p0_1x, clk_p0_4x);
endmodule
```

See also "Using Compiler Directives to Manage Out-of-Module References to SPICE" on page 189.

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'bx`) 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'bx`.

```
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 a Command-Line Option to Manage Out-of-Module References to SPICE” on page 188.

Instantiating Verilog-AMS and VHDL-AMS in SystemC

Virtuoso AMS Designer provides support for instantiating Verilog (digital), Verilog-AMS, VHDL (digital), and VHDL-AMS modules inside SystemC® models.

SystemC is a digital-only language. For that reason, AMS Designer 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, AMS Designer uses connect modules and interface modules.

When using SystemC models with the AMS Designer simulator, the following restrictions apply:

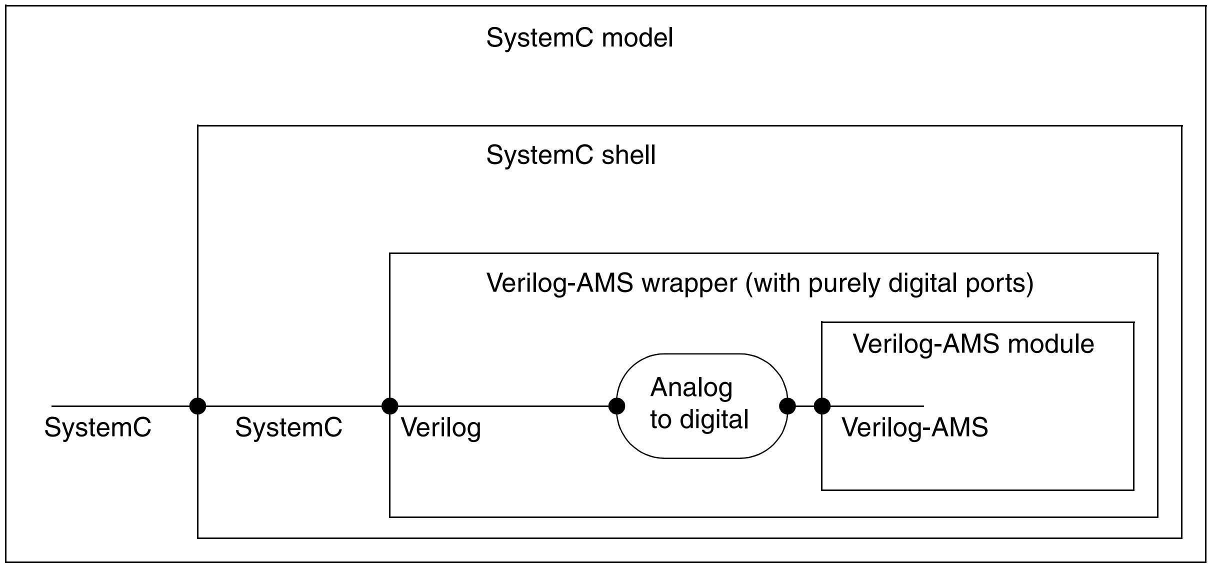
- You must not instantiate a Verilog-AMS module that instantiates a VHDL-AMS module
- You must not instantiate a VHDL-AMS module that instantiates a Verilog-AMS module

See the following topics for more information:

- Preparing and Using Wrappers and Shells on page 192
- Preparing Interface Modules on page 194
- Guidelines for Using AMS Modules in SystemC Models on page 197

Preparing and Using Wrappers and Shells

As the following diagram illustrates, AMS Designer 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.

Syntax of the ncshell Command

You can create the Verilog-AMS wrapper and the SystemC shell with the `ncshell` command. For importing Verilog-AMS modules, the command is:

```
ncshell_command ::=
        ncshell {ncshell_param} -ams -import verilog -into systemc
            lib.module {analogim_param}

analogim_param ::=
        -analogim child_port [@interface_mod_port]
        :lib2.interface_mod[:view]
```

For importing VHDL-AMS modules, the command is:

```
ncshell_command ::=
        ncshell {ncshell_param} -ams -import vhdl -into systemc
            lib.module {analogim_param}
```

```
analogim_param ::=
        -analogim child_port [@interface_mod_port]
        :lib2.interface_mod[:view]
```

`ncshell_param`	An additional `ncshell` option that you might wish to use. For a list of available options, use `ncshell -help`. For example, you might use the `-messages` or `-sctype` options.
`lib`	Library containing the Verilog-AMS or VHDL-AMS module that is to be imported into the SystemC model.
`module`	Verilog-AMS or VHDL-AMS module that is to be imported into the SystemC model.
`-analogim`	Parameter indicating that *`lib2.interface_mod`* is to be used as the interface module for *`child_port`*.
`child_port`	Port of the imported Verilog-AMS or VHDL-AMS for which the *`interface_mod`* module is inserted. If *`child_port`* is specified as `*`, the *`interface_mod`* module is inserted in the Verilog-AMS wrapper for each port of *`module`*.
`interface_mod_port`	Name of the interface module port to be bound to *`child_port`*. If *`interface_mod_port`* is not specified, the `ncshell` program uses whichever of the two ports of *`interface_mod`* is compatible with *`child_port`*: the analog port is connected to child ports declared as analog; otherwise, the discrete port is connected.
`lib2`	Library containing *`interface_mod`*.
`interface_mod`	The interface module to be used to translate analog signals from the Verilog-AMS or VHDL-AMS module into digital signals, or vice versa.
`view`	View of *`interface_mod`* to be used as the interface module.

The wrapper generated by this `ncshell` 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.

Examples of the ncshell Command

The following examples illustrate different ways of using the `ncshell` 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).

```
ncshell -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`.

```
ncshell -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
```

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 `ncshell` 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 port and the net must appear in the same row of the following table.

SystemC Net	Description	Verilog Equivalent	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 unsigned int	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 fixed point	logic [W-1 : 0]	analog [W-1 : 0]
sc_unfixed <W,I,Q,O,N>	templated unsigned fixed 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 `ncshell` command such as the following

```
ncshell -ams -mess -import verilog -into systemc -sctype p1: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 `ncshell` 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 `ncshell` program to pass parameters to interface modules.

- SystemC models offer limited compatibility checking.

  The `ncshell` program performs only limited checking on the compatibility of interface module ports and imported module ports. The `ncshell` program does not check data type or discipline compatibility.

- Discipline resolution results are not available to the `ncshell` program.

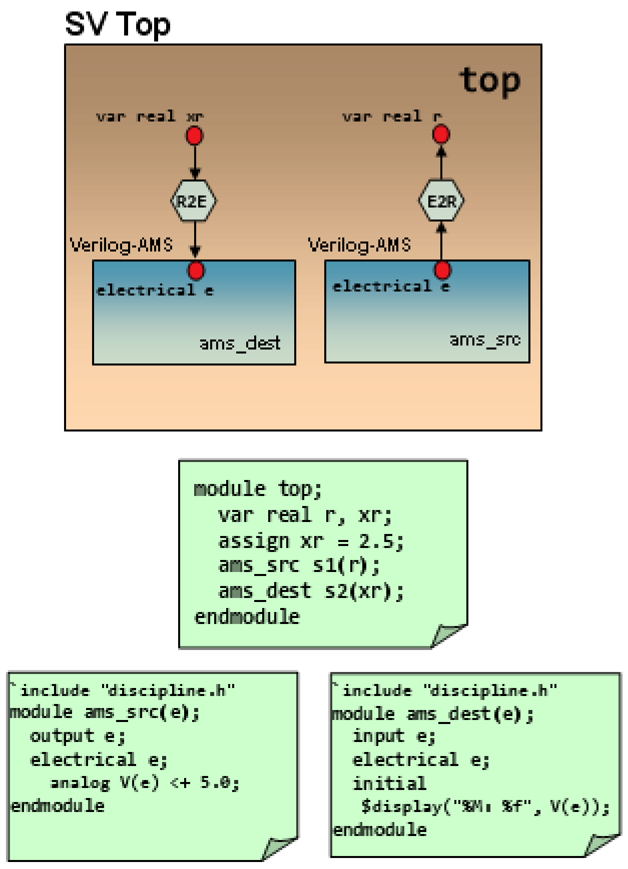
  Ports declared as `wire` in imported modules can resolve to either continuous or discrete after discipline resolution. The `ncshell` program, however, is used prior to discipline resolution and so the ultimate discipline of ports declared as `wire` is unknown. The `ncshell` program assumes that a wire is non-analog and connects the non-analog port of the interface module to it. This choice might, or might not, be correct.

Using SystemVerilog Modules

When simulating designs that contain SystemVerilog modules together with Verilog-AMS and VHDL-AMS modules, the following guidelines apply:

- You may use either single-step (`irun`) or three-step (`ncvlog`, `ncelab`, `ncsim`) simulation method to run a design with both SystemVerilog and AMS blocks.
  When you use the `irun` 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 `irun` commandline 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 via discipline resolution if they are 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. Refer to Figure 7-2 on page 199 and Figure 7-3 on page 200 for an examples.

Figure 7-2 SystemVerilog on top instantiating Verilog-AMS: electrical port in Verilog-AMS

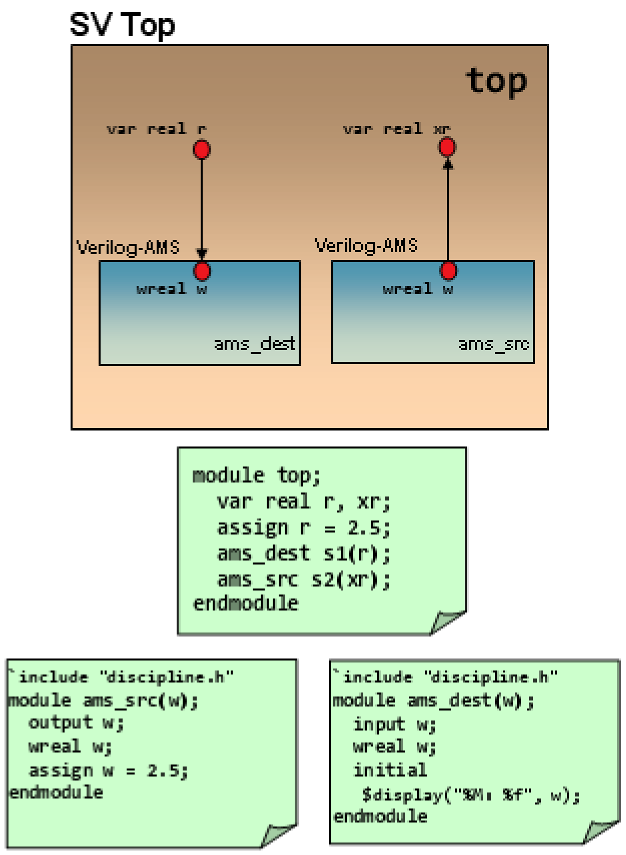


```
module top;
  var real r, xr;
  assign xr = 2.5;
  ams_src s1(r);
  ams_dest s2(xr);
endmodule
```

```
`include "discipline.h"
module ams_src(e);
  output e;
  electrical e;
    analog V(e) <+ 5.0;
endmodule
```

```
`include "discipline.h"
module ams_dest(e);
  input e;
  electrical e;
  initial
   $display("%M: %f", V(e));
endmodule
```

Figure 7-3 SystemVerilog on top instantiating Verilog-AMS: wreal port in Verilog-AMS



```
module top;
  var real r, xr;
  assign r = 2.5;
  ams_dest s1(r);
  ams_src s2(xr);
endmodule
```

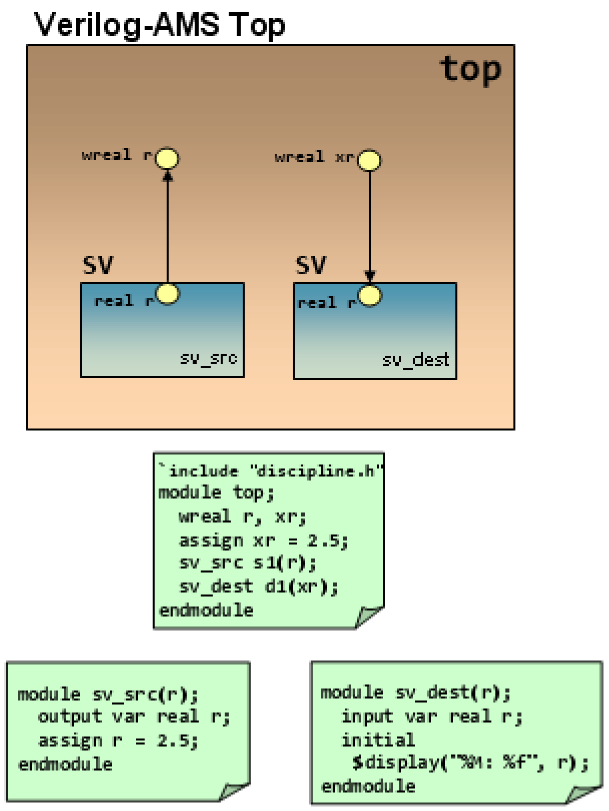
```
`include "discipline.h"
module ams_src(w);
  output w;
  wreal w;
  assign w = 2.5;
endmodule
```

```
`include "discipline.h"
module ams_dest(w);
  input w;
  wreal w;
  initial
   $display("%M: %f", w);
endmodule
```

Note: Power-smart 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 (Refer to Figure 7-4 on page 201 for an example). 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. (Refer to Figure 7-5 on page 202 for an example).

Figure 7-4 Verilog-AMS on top instantiating SystemVerilog: wreal port in Verilog-AMS

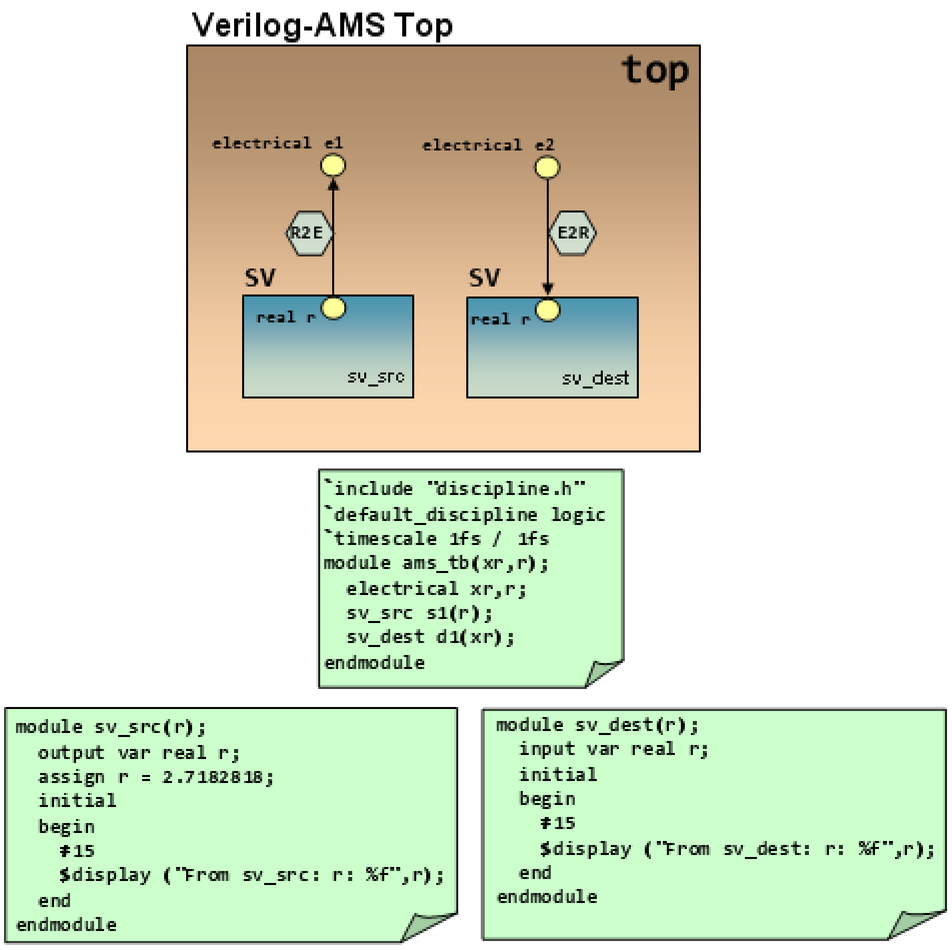


```
`include "discipline.h"
module top;
  wreal r, xr;
  assign xr = 2.5;
  sv_src s1(r);
  sv_dest d1(xr);
endmodule
```

```
module sv_src(r);
  output var real r;
  assign r = 2.5;
endmodule
```

```
module sv_dest(r);
  input var real r;
  initial
   $display("%M: %f", r);
endmodule
```

Figure 7-5 Verilog-AMS on top instantiating SystemVerilog: electrical port in Verilog-AMS



```
`include "discipline.h"
`default_discipline logic
`timescale 1fs / 1fs
module ams_tb(xr,r);
  electrical xr,r;
  sv_src s1(r);
  sv_dest d1(xr);
endmodule
```

```
module sv_src(r);
  output var real r;
  assign r = 2.7182818;
  initial
  begin
    #15
    $display ("From sv_src: r: %f",r);
  end
endmodule
```

```
module sv_dest(r);
  input var real r;
  initial
  begin
    #15
    $display ("From sv_dest: r: %f",r);
  end
endmodule
```

- 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

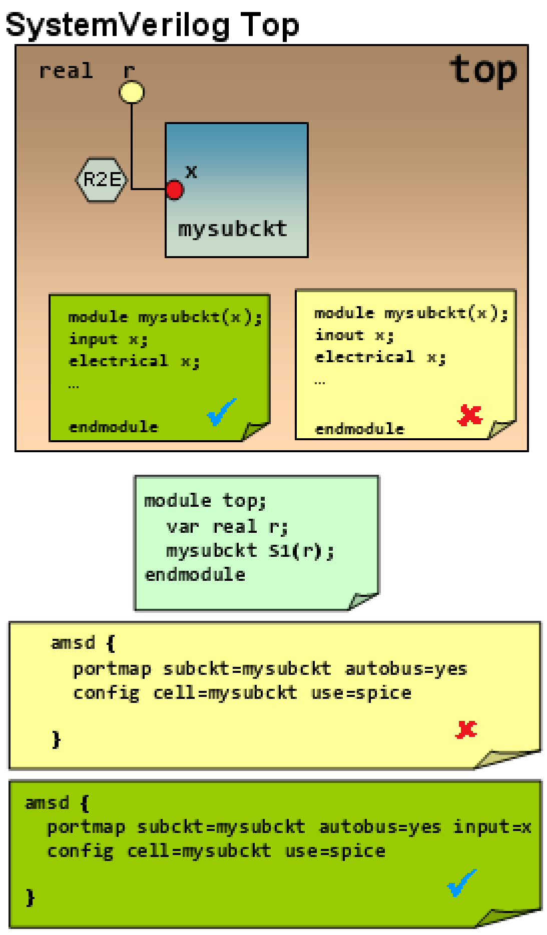
    Note: See "Including Subcircuits and Models" on page 150 for more information.

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. Refer to Figure 7-6 on page 204 for an example.

Figure 7-6 SystemVerilog on top, SPICE subckt Underneath



```
module mysubckt(x);
input x;
electrical x;
...
endmodule
```

✓

```
module mysubckt(x);
inout x;
electrical x;
...
endmodule
```

✗

```
module top;
  var real r;
  mysubckt S1(r);
endmodule
```

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

}
```

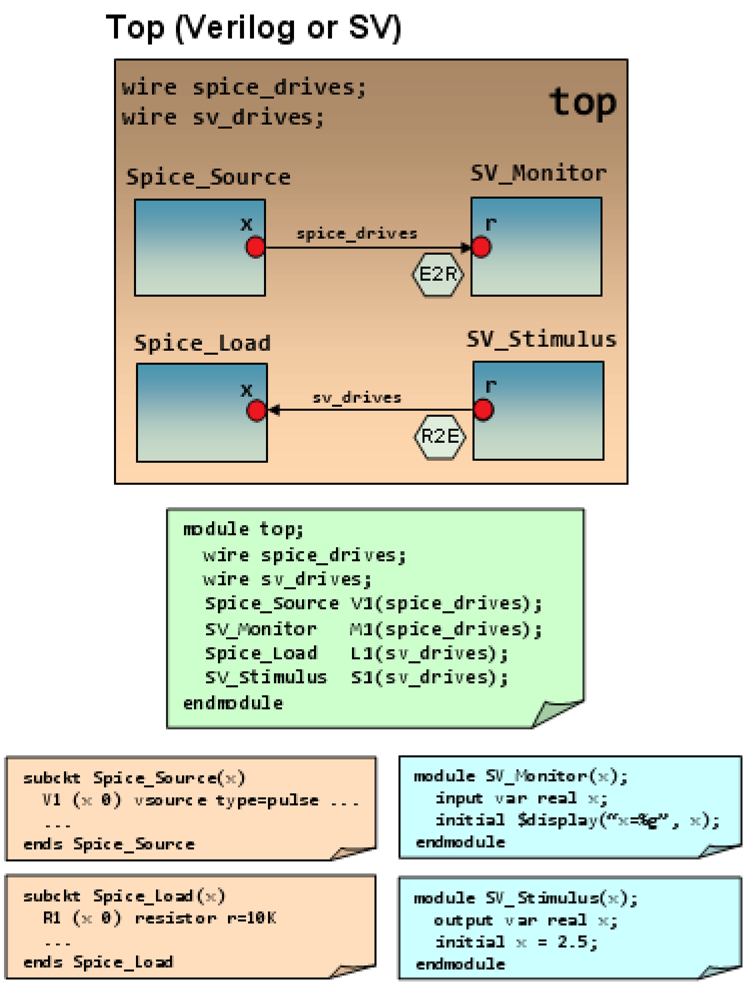
✗

```
amsd {
  portmap subckt=mysubckt autobus=yes input=x
  config cell=mysubckt use=spice

}
```

✓

- 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. Refer to Figure 7-7 on page 205 for an example.

Figure 7-7 Side-by-side connection between SPICE and SystemVerilog blocks



```
wire spice_drives;
wire sv_drives;
```

```
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
```

Applying Assertions to real, wreal, and electrical Nets

The AMS Designer simulator lets 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.

Note: PSL stands for the property specification language that is undergoing standardization as IEEE 1850 Property Specification Language.

See the following topics for details:

- SystemVerilog Assertions on page 206
- Using Analog System Tasks in SVA on page 208
- PSL Assertions on page 209
- Limitations of Using PSL Assertions on page 210

For more information on writing assertions, refer to the *Assertion Writing Guide*.

SystemVerilog Assertions

Using the AMS Designer simulator, 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.

Note: 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. See Using SystemVerilog Modules on page 198 for details.

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_s1(r);  // causes insertion of Electrical2Real connection
                               // module
  ams_electrical_dst e_d1(xr); // causes insertion of Real2Electrical connection
                               // module
  ams_wreal_src      w_s1(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
```

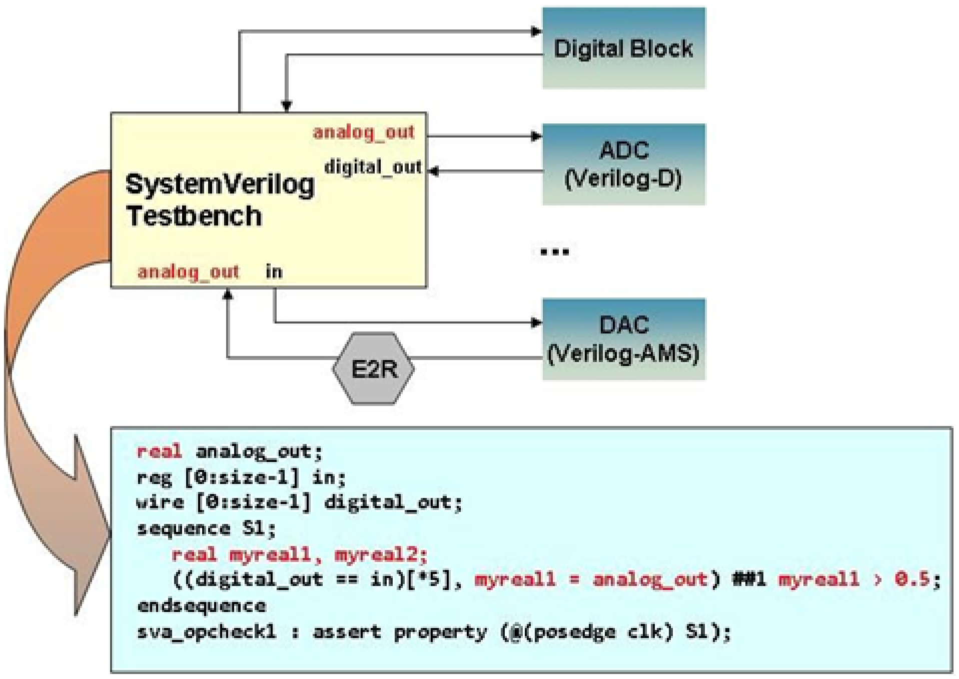
Once the SystemVerilog real variables import the AMS functionality through the code above, they can appear in any SystemVerilog assertion statement that permits the use of real variables. For more details on how real variables can be used in SystemVerilog assertion, refer to *SVA Quick Reference*.

Here is an example of specifying an assertion on a SystemVerilog `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 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
```

Figure 7-8 on page 208 shows another example of a SystemVerilog real variable being used inside a SystemVerilog assertion.

Figure 7-8 SystemVerilog real variable used inside a SystemVerilog assertion



```
real analog_out;
reg [0:size-1] in;
wire [0:size-1] digital_out;
sequence S1;
   real myreal1, myreal2;
   ((digital_out == in)[*5], myreal1 = analog_out) ##1 myreal1 > 0.5;
endsequence
sva_opcheck1 : assert property (@(posedge clk) S1);
```

Using 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.

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 allows you to use PSL assertions with analog and mixed-signal blocks that you model using the Verilog-AMS `wreal` data type.

Important

You must use the `-assert` option with the `irun` or `ncvlog` 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.

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
```

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.

Using Common Power Format with AMS Designer

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. 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 AMS Designer simulator provides support for Common Power Format (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_power_mode`
- `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`
- `define_level_shifter_cell`
- `update_level_shifter_rules`

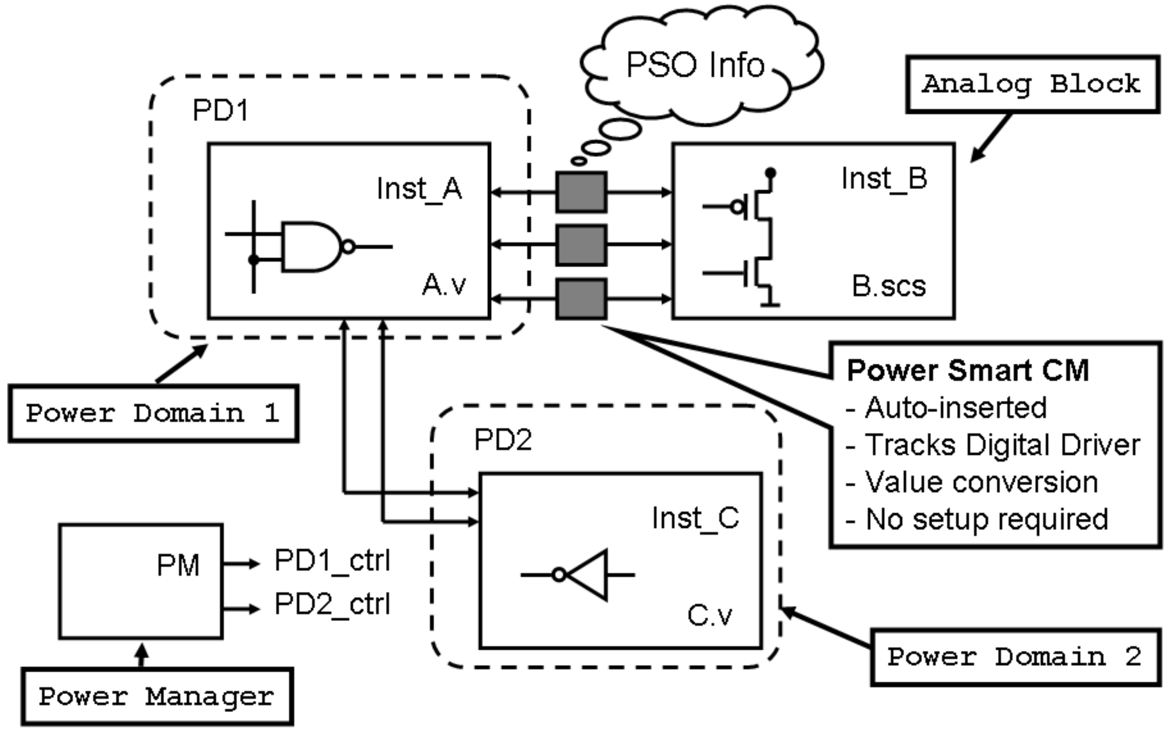
CPF is a standard format, based on Tcl, for specifying all power-specific information and constraint requirements from design through verification and implementation.

See the following documents for detailed information about CPF and CPF commands:

- *Common Power Format User Guide*
- *Common Power Format Language Reference*
- *Low-Power Simulation Guide*

Power-Smart Connect Modules

The AMS Designer simulator 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 "CPF"). The power-smart connect modules 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. 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 AMS Designer simulator 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 AMS Designer simulator 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.

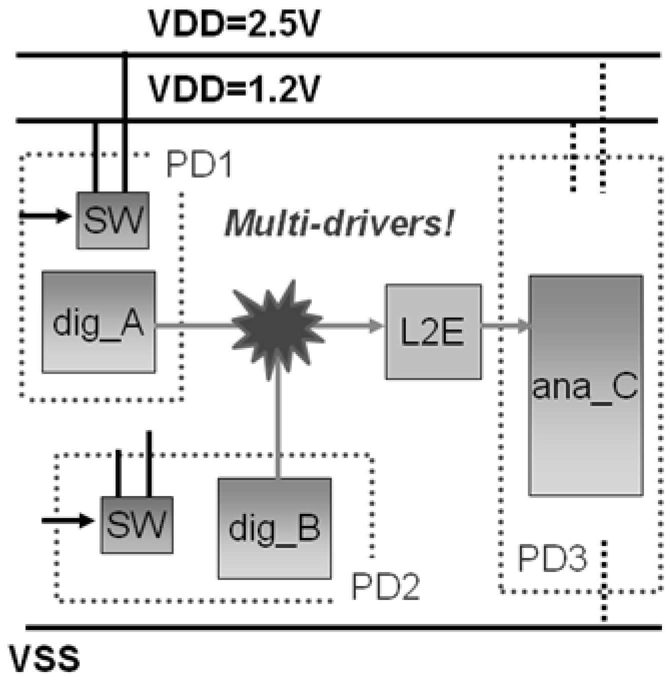
Also, the power-smart connect modules support advanced power-domain features like power modes and nominal conditions, resulting in improved digital-to-analog transition inside the modules in low power mixed-signal simulation. A power-smart connect module in AMS-CPF can:

- 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.

Note: Refer to the connect library in the installation to see the power-smart connect module definitions.

Support for Multiple Digital Drivers

Let us 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 `vpso`, 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_CPF__ddiscrete_2_4.Din), only one will be used :
PowerDomain : PD_Dig1  - Voltage : 1.800000 (selected)
PowerDomain : PD_Dig2  - Voltage : 1.600000
```

Using the ie Card in AMS-CPF

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. The `vpso` value overrides the `vx_amscpf_poweroff` parameter in the power-smart connect module.

Example:

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

Using Power Aware Modeling

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();
```

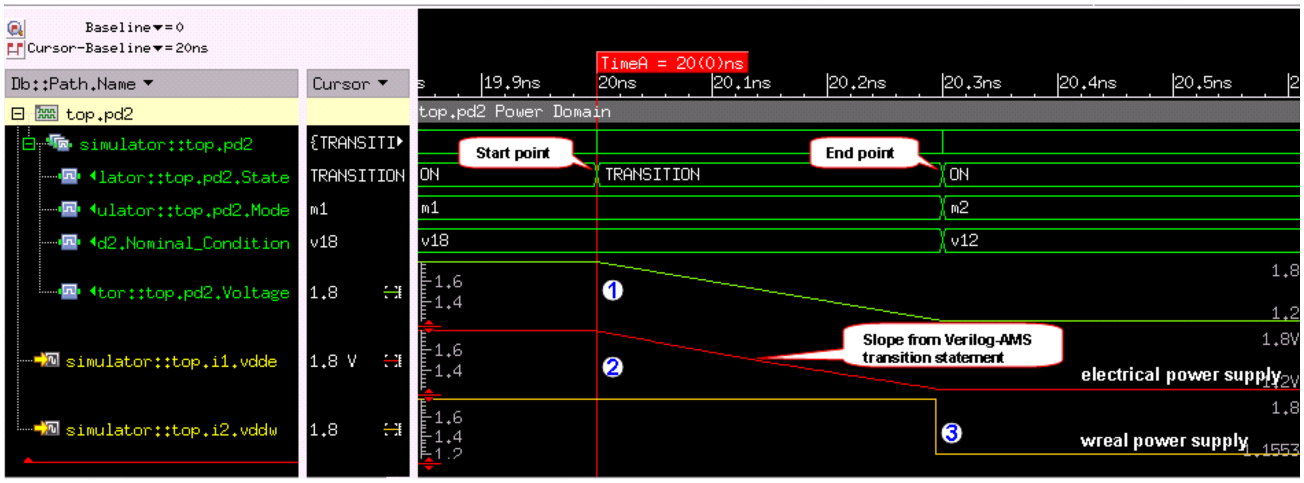
To obtain more information about these system tasks, refer to the *Power-Aware Modeling* chapter of the *Low-Power Simulation Guide*.

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
```

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 specification. The power supply value changes at the start point of 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 end point of the transition, which enables you to determine whether the wreal power supply is in the transition period.



Note: If the power supply net is specified on a VHDL net, the net is removed from the auto supply net list. In addition, 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 AMS Designer simulator 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.

Note: This feature is enabled by specifying the `-lps_ams_lsr` command-line option.

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 AMS Designer simulator 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.
- A power-smart IE connects to the digital loads with different power domains.

  Note: You can use the `-lps_ams_relax_pdchk` command-line option to direct the simulator to generate a warning instead of error for the above scenario.
- A power-smart IE connects to the analog signals (drivers or loads) with different power domains.
- A power-smart IE connects to wreal or real signals (drivers and loads) with different power domains.

Note: The simulator does not generate an error if the digital driver and load belong to different power domains.

This feature works only when the nominal condition is specified in the CPF file and enabled for low power simulation. You must specify the following:

- `-lps_pmode` or `-lps_mvs`
- `-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:

- `create_nominal_condition`
- `create_power_domain -active_state_condition` or `create_power_mode`

Using wreal Data Type

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, following two options are supported:

- `-lps_wreal_nocorrupt`

  It prevents the forcing of the wreal signal and also prevents the original value from getting corrupted.
- `-lps_wreal_corrupt_value <value>`

This option is provided to make it 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`* `]`

  This option specifies that the objects of a power domain are non-corrupt during simulation and maintains their values.

- `wreal_corrupt_value [` *`value`* `|` *`value@object`* `]`

  This option specifies the corruption value of objects in the power domain.

Support for 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 (`ncelab`), 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 connect with the digital side of connect module during driver-receiver segregation.

During the simulation phase (`ncsim`), AMS-CPF checks the power status of the boundary port domain in the `amscpf-vpi` function.

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`

Note: 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 !==`

  Note: 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
  ```

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 wrl
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} }
```

Note: Wreal instances that are specified using the `create_power_domain` CPF command also have the power corruption value applied to them.

Support for 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 IUS11.1, 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*.

Fetching Values Associated with an Analog Object

You can use the value fetch 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 continuous domain. It can be called from within Verilog, SystemVerilog, 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 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.

  Note: 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 index can be variable, reg, or parameters so long as their value evaluates to an integral constant.

  Note: Index applies only for vector objects.

- 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.

Using the 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 can be turned on by using the `ncelab` or `irun -amssie` command-line option.

To use the `-amssie` option, all connect modules inserted in the design must support SIE, otherwise, the simulator will generate an error. It is recommended that you configure your designs 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 `irun` command-line flow, use the `ie` statement in the `amsd` block. For example, `ie connrules=<`*`conn_rule_name`*`>`.
- In the `ncelab` or `irun` command-line flow, specify the connect rules. For example, `ConnRules_18V_full_fast`.

The following new set of parameters for corresponding strength resistances are defined in the `full_fast` connect rules:

Parameter Name	Default Value
`rsupply`	`4`
`rout`	`200`
	For default strength, which is `strong`
`rlo`, `rhi`	`rout`
	For `strong0` and `strong1` respectively
`rpull`	`1.5e3;`
`rlarge`	`9.0e3;`
`rweak=`	`5.5e4;`
`rmid`	`3.2e5;`
`rsmall`	`1.9e6;`
`rz`	`1.0e7;`

Note: 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 on page 120). 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.

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 flow.

8

Real Number Modeling using the AMS Designer Simulator

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).

Note: 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.

See the following topics for more information:

- Using Wreal Concatenation Expressions on page 267
- Creating L2R and R2L Connect Modules on page 268
- Using Virtuoso Visualization and Analysis in irun and ADE Flows for Simulations with Real Number Models on page 277

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 `IUS_INST_DIR/tools/amsd/wrealSamples` area. These libraries contain text models and also the versions that can be used with the IUS5141 and IC61x releases. The text models are directly accessible and usable in a command-line digital-centric flow.

Note: For details about the `wreal` data type, see "Real Nets" in the "Data Types and Objects" chapter of the Cadence® Verilog-AMS Language Reference.

See the following topics for information about `wreal` features of the AMS Designer simulator:

- Basic wreal Features of the AMS Designer Simulator on page 232
- Advanced wreal Features of the AMS Designer Simulator on page 233

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
  ```

  For more information, see "Arrays of Real Nets" in the *Cadence® Verilog®-AMS Language Reference*.

- `wreal` nets with more than one driver

  For more information, see "Real Nets with More than One Driver" in the *Cadence® Verilog®-AMS Language Reference*. See also "Selecting a wreal Resolution Function" on page 244.

- `wrealXState` and `wrealZState`

  For more information, see "Real Nets with More than One Driver" in the *Cadence® Verilog®-AMS Language Reference*.

- Connections between `wreal` and VHDL `real` signals

  See also "Connecting VHDL and VHDL-AMS Blocks to Verilog and Verilog-AMS Blocks" on page 163.

- `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  ;
end
assign y = yval;
endmodule
```

Note: You can use irun/ncelab option `-rnm_coerce` with parameters to turn ON or OFF the wreal coercion feature. The default is `ON` for whole design.

See also "Instantiating VHDL Blocks with Real Signal Ports on a Schematic" on page 243 and "Connecting VHDL and VHDL-AMS Blocks to Verilog and Verilog-AMS Blocks" on page 163.

- `wreal` nets with inherited connections

  This includes:

  - Connection between a `wreal` net type and another `wreal` net type through an inherited connection.
  - Connection between a `wreal` net type and an `electrical` net type 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.

  Note: Connection between the `wreal` net type and the `real` net type is not supported.

  Example (`wreal` net type with another `wreal` net type):

```
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

  For more information, see "Selecting a wreal Resolution Function" on page 244.

- `wreal` as an independent variable in the `$table_model` function

  For more information, see "Using wreal Independent Variables in a $table_model" on page 237.

- 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;
    ```

Using 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.

Note: In order to use the $table_model function in a digital context, you must be using digital mixed-signal licensing or AMS Designer simulator licensing according to the “Feature-to-License Checklist” on page 27.

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.

Note: For more information about the `$table_model` function, see "Interpolating with Table Models" in the *Cadence Verilog-AMS Language Reference*.

Connecting Verilog-AMS wreal Signals to Analog Signals

In AMS-Spectre, 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 will find the E2R and R2E connect modules in the Cadence hierarchy at

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
```

Note: In addition to connecting Verilog-AMS wreals to electrical nets, you can connect the SystemVerilog real ports to electrical nets.

Resolving Disciplines for Verilog-AMS wreal Nets

The software 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.

Just as with Verilog-AMS wires, `wreal` nets can have any discipline (discrete or continuous) based on how they are declared, used, or resolved.

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`).

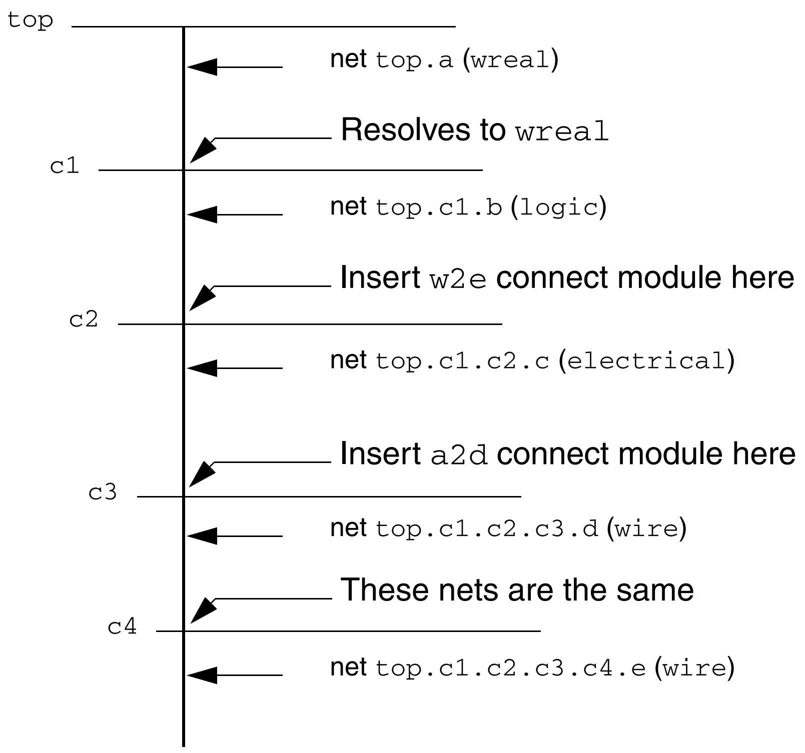
Nets `top.c1.b` (discrete) and `top.c1.c2.c` (continuous) have incompatible disciplines (`logic`/`wreal` versus `electrical`) as do nets `top.c1.c2.c` and `top.c1.c2.c3.d` (`electrical` versus `logic`/`wire`) so the software needs to insert connect modules where these nets connect. The software determines 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.

```
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
```


top
net top.a (wreal)
Resolves to wreal
c1
net top.c1.b (logic)
Insert w2e connect module here
c2
net top.c1.c2.c (electrical)
Insert a2d connect module here
c3
net top.c1.c2.c3.d (wire)
These nets are the same
c4
net top.c1.c2.c3.c4.e (wire)

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 also "Connecting Verilog-AMS wreal Signals to Analog Signals" on page 239.)

Using wreal Nets at Mixed-Language Boundaries

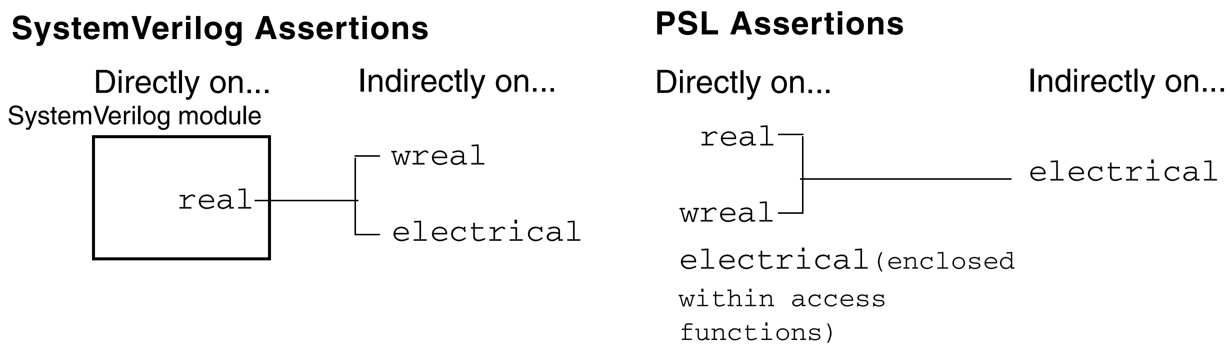
You can use `wreal` nets at mixed-language boundaries. However, if you have any Verilog-AMS `wreal` nets driving a VHDL real number receiver, that receiver must be able to handle the `wrealZState` and `wrealXState` state values. The software represents these values as double-precision (64-bit) NaNs (Not-a-Number) 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.

Note: See *`your_install_dir`*`/tools/affirma_ams/etc/connect_lib/README` for detailed information about the set of connect rules that Cadence provides.

See also "Selecting a wreal Resolution Function" on page 244 for information about using the `-wreal_resolution` command-line option during elaboration to select a `wreal` resolution function.

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.



Note: For more information, refer to Applying Assertions to real, wreal, and electrical Nets on page 206.

Instantiating 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.

Note: AMS Designer resolves Verilog-AMS `wreal` nets to the analog or digital domain during the discipline resolution process (see "Resolving Disciplines for Verilog-AMS wreal Nets" on page 240). 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.

Selecting a wreal Resolution Function

You can define a resolution function for each net in a design on per scope basis by using the Verilog-AMS discipline mechanism. You can also designate a global resolution function that will be used for undisciplined nets and for the nets with disciplines that do not define a `wreal` resolution function.

Defining a wreal Resolution Function for a Discipline

There are two methods for defining `wreal` resolution functions for a discipline.

- Defining a Resolution Function Directly in the Discipline Definition on page 244
- Defining a Resolution Function with the AMSCB connectmap Card on page 245

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`* can have any of the following values:

- default on page 248
- fourstate on page 249
- sum on page 251
- avg on page 252
- min on page 253
- max on page 254
- `none`

Note: If you specify the value of *`resolveFunction`* as `none`, it is equivalent to not using the `realresolve` parameter at all. Nets of such disciplines will use the global resolution function. An error will occur 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;
enddisicipline
```

Important

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.

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.

For more information about the `connectmap` card, its syntax, and examples, see connectmap on page 132.

Defining a Global wreal Resolution Function

There are two methods that can be used to define a global `wreal` resolution function:

- Using the -wreal_resolution Command-Line Option to ncelab and irun on page 245
- Using the AMSCB connectmap Card on page 246

The resolution function defined through the `-wreal_resolution` command-line option will have 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 will take its value from the command-line option and a warning will be printed on the screen.

If no global `wreal` resolution function is specified using these two methods, it will be set to the `default` `wreal` resolution function.

Using the -wreal_resolution Command-Line Option to ncelab and irun

You can use the `-wreal_resolution` command-line option with `ncelab` or `irun` 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 following:

- default on page 248
- fourstate on page 249
- sum on page 251
- avg on page 252
- min on page 253
- max on page 254

If you do not specify a resolution function using this command-line option, the elaborator uses the `default` resolution function algorithm.

Note: For those cases where the elaborator issues a runtime warning message, you can cause the message to be fatal using the `-ncfatal` command-line option. For more information about this option, look for `-ncfatal` in "Compiling Verilog Source Files with ncvlog" in *Cadence Verilog Simulation User Guide*.

Caution

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`***.***

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`.

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. Resolution function explicitly defined in the AMS Control Block using the `connectmap` card
2. Resolution function explicitly defined in the discipline definition
3. Resolution function of the discipline of the net to which a given net is connected
4. Global resolution function specified with the `-wreal_resolution` command-line option
5. Global resolution function specified with the `connectmap` card (without the `discipline` parameter)
6. The `default` global resolution function

Determining 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 will be that resolution function.

  Note: 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 will occur.

- 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 will be 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

Reporting the Resolution Functions of wreal Nets

You can use the `+wreal_res_info` argument with `ncelab` and `irun` to print a report of all `wreal` nets in the design and their resolution functions.

Predefined wreal Resolution Functions

The following predefined `wreal` resolution functions can be associated with 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 `wrealZState`	Drive the receivers using `wrealZState`

Conditions	Resolution
Exactly one driver is not driving `wrealZState`	Drive the receivers using the only non-`wrealZState` value
More than one driver is not driving `wrealZState`	Drive the receivers using `wrealXState` and issue a runtime warning message
Any driver is driving `wrealXState`	Drive the receivers using `wrealXState`

Here is an example of how the elaborator determines the resolved value of two drivers, `wreal1` and `wreal2` using the `default` 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	`wrealXState`[1]
1.23	3.33	`wrealXState`[1]
1.23	`wrealXState`	`wrealXState`
`wrealXState`	3.33	`wrealXState`
`wrealXState`	`wrealZState`	`wrealXState`

1. If there is more than one non-`wrealZState` driver, the software issues a runtime warning message.

See also "Real Nets with More than One Driver" in the *Cadence® Verilog®-AMS Language Reference* for information about the `wrealXState` and `wrealZState`.

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 `wrealZState`	Drive the receivers using `wrealZState`
Exactly one driver is not driving `wrealZState`	Drive the receivers using the only non-`wrealZState` value
More than one driver is not driving `wrealZState`	■ When all non-`wrealZState` drivers drive the same value, drive the receivers using that value and issue a runtime warning message ■ When all non-`wrealZState` driver do not drive the same value, drive the receivers using `wrealXState` and issue a runtime warning message
Any driver is driving `wrealXState`	Drive the receivers using `wrealXState`

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
`wrealZState`	`wrealZState`	`wrealZState`
1.23	`wrealZState`	1.23
`wrealZState`	3.33	3.33
1.23	1.23	1.23
1.23	3.33	`wrealXState`[1]
1.23	`wrealXState`	`wrealXState`
`wrealXState`	3.33	`wrealXState`
`wrealXState`	`wrealZState`	`wrealXState`

1. If there is more than one non-`wrealZState` driver, the software issues a runtime 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 `wrealZState`
Any driver is driving `wrealZState`	Drive the receivers with the summation of all non-`wrealZState` values (that is, ignore all `wrealZState` drivers)
No driver is driving `wrealZState`	Drive the receivers with the summation of all the drivers
Any driver is driving `wrealXState`	Drive the receivers using `wrealXState`

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
`wrealZState`	`wrealZState`	`wrealZState`
1.23	`wrealZState`	1.23
`wrealZState`	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 `wrealZState`	Drive the receivers using `wrealZState`
Any driver is driving `wrealZState`	Drive the receivers with the average of all non-`wrealZState` values (that is, ignore all `wrealZState` drivers)
No driver is driving `wrealZState`	Drive the receivers with the average of all the drivers
Any driver is driving `wrealXState`	Drive the receivers using `wrealXState`

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
`wrealZState`	`wrealZState`	`wrealZState`
1.23	`wrealZState`	1.23
`wrealZState`	3.33	3.33
1.23	1.23	1.23
1.23	3.33	2.28
1.23	`wrealXState`	`wrealXState`
`wrealXState`	3.33	`wrealXState`
`wrealXState`	`wrealZState`	`wrealXState`

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 `wrealZState`
Any driver is driving `wrealZState`	Drive the receivers with the minimum of all non-`wrealZState` values (that is, ignore all `wrealZState` drivers)
No driver is driving `wrealZState`	Drive the receivers with the minimum of all the drivers
Any driver is driving `wrealXState`	Drive the receivers using `wrealXState`

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
`wrealZState`	`wrealZState`	`wrealZState`
1.23	`wrealZState`	1.23
`wrealZState`	3.33	3.33
1.23	1.23	1.23
1.23	3.33	1.23
1.23	`wrealXState`	`wrealXState`
`wrealXState`	3.33	`wrealXState`
`wrealXState`	`wrealZState`	`wrealXState`

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 `wrealZState`
Any driver is driving `wrealZState`	Drive the receivers with the maximum of all non-`wrealZState` values (that is, ignore all `wrealZState` drivers)
No driver is driving `wrealZState`	Drive the receivers with the maximum of all the drivers
Any driver is driving `wrealXState`	Drive the receivers using `wrealXState`

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`

Using 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 net types (discrete time domain) (see Writing RNMs Using Disciplineless Wreal Nets)
- Portable Verilog-AMS wreal models for use in SystemVerilog (SV) (see Writing Portable Verilog-AMS Wreal Models for Use in SV on page 256)

This capability enables you to reuse your code according to the changes to the SV modeling features.

Writing RNMs Using Disciplineless Wreal Nets

You can resolve multiple drivers on wreal signals to suit the expected changes in SV. To accomplish this, you can use the six new disciplineless wreal net types—wrealavg, wrealsum, wrealmax, wrealmin, wreal1driver, and wreal4state—added to Verilog-AMS.

These disciplineless wreal net types enable you to write wreal models that align with the SV-DC requirement.

Example:

To declare disciplineless wreal nets that have resolution associated to them, write:

```
    wrealsum wr_sum_1;
    wrealavg wr_avg_1;
    wrealmin wr_min_1;
```

Here is a full code example using the new wreal net types:

```
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
```

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 once the new LRM is released. The use model is as follows:

1. Create wreal models in Verilog-AMS using the new disciplineless net types.
2. Once the IEEE 1800-2012 standard is released and implemented, 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 net types that correspond to the current disciplineless wreal net types.

   Note: 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 net types, for example wrealsum, for multiple driver resolution function.

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
```

Using Real Number Modeling in SystemVerilog

You can enable real number modeling in SystemVerilog by creating a set of built-in net types with real data type and built-in resolution functions equivalent to the wreal resolution functions.

A built-in real SV net type can be declared as follows:

```
nettype real nettype_identifier with builtin_res_func;
```

Where:

`nettype_identifier` is the identifier that you use for the net type 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 net type, 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 shown in the following table:

Table 8-1 Built-In Resolution Functions

Built-in Resolution Function	Equivalent Wreal Type
`CDS_res_wreal1driver`	`wreal1driver`
`CDS_res_wreal4state`	`wreal4state`
`CDS_res_wrealmin`	`wrealmin`
`CDS_res_wrealmax`	`wrealmax`
`CDS_res_wrealsum`	`wrealsum`
`CDS_res_wrealavg`	`wrealavg`

Once a built-in net type has been declared, a net of the built-in net type can be used just like a net of any other user-defined net type as governed by the SV LRM.

```
nettype real wrealavg with CDS_res_wrealavg;  //declare a built-in net type
                                                "wrealavg"
wrealavg x;           //declare a singular net of net type "wrealavg"
wrealavg x [0:3];    // declare a 4-element array of nets of net type "wrealavg"
```

You can easily port existing wreal models to SV by using the `cds_rnm_pkg` package that defines a set of built-in net types that are equivalent to the typed wreal nets. For example, a 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 net types 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 IUS installation at `$INSTALL_ROOT/tools/affirma_ams/etc/dms/cds_rnm_pkg.sv`.

You can use the `irun -rnm_package` command-line option to automatically append the `cds_rnm_pkg` package to the `irun` command line. This builds the package and makes it available for import in all SV modules that irun is compiling.

Resolving Wreal Nets of built-in Net Type

The resolution function of wreal nets of built-in net type runs at time 0 regardless of the state of its drivers to bring them into compliance with the SystemVerilog nets of net type, as described in the IEEE 1800-2012 standard.

Note: In releases prior to 13.1, these nets initialized to 0.0 and resolved to a new value only when the value of one of their drivers changed.

Consider the following example:

```
module top();
    wreal1driver x;

    real a;
    real b;

    initial
```

```
        begin
            a = 0;
            b = 0;
            #1 a = 1.1;
            #1 b = `wrealZState;
            #1 $finish();
        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 net type `wreal1driver` and has two drivers `a` and `b` both driving 0.0 at time 0.

Running the above example results in the following output:

```
ncsim: *W,WRMNZD: top.x: wreal net has multiple non-Z drivers (Time: 0 NS + 1).
                   0: X = `wrealXState
ncsim: *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
./top.vams:13   #1 $finish();
```

In the above output, `x` resolves to `wrealXState` at time 0.

Handling Port Connections

Port Connections of nets of built-in net type will be governed by the port connection rules in the LRM (section 23.3.3).

Both the upper and lower connections are nets of built-in net type

- If both are singular nets of built-in net type then they must have a matching net type as governed by the matching net type 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 net type, their elements must have matching net type as described in section 6.22.6 of the LRM. The port shall be merged into a single simulated net.

Note: You can use the ncelab/irun `-nettype_port_relax` command-line option to allow for relaxed port compatibility rules for connections of built-in net types. If this option is specified the following rules will govern connections of built-in net type to built-in net type:

- 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 net types or wreals. In addition, the resolution functions of their element net types must be the same. The port shall be merged into a single simulated net.

One is a net of built-in net type and the other is a wreal net

- If both are singular nets then the resolution function of the net of built-in net type 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 net type of the net of built-in net type must match the wreal type from Table 8-1 on page 257. The port shall be merged into a single simulated net.

Note: 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 net type and the other is a variable or expression

- If the net of built-in net type 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 net type 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.

Wreal Interaction With Nets of Built-In Net Type

Wreal Coercion

In general, nets of built-in net type 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 on page 259 is considered.

Connecting an implied interconnect wire to a net of built-in net type will coerce that implied interconnect to a wreal net. The type of the coerced wreal net will be determined by identifying the wreal type that corresponds to the net of built-in net types resolution functions described in Table 8-1 on page 257.

Connecting a net of built-in net type to a generic wreal net will coerce the type of the generic wreal net in the same manner that it coerces an implied interconnect. The final type of the generic wreal will be determined by identifying the wreal type that corresponds to the net of built-in net types resolution function described in Table 8-1 on page 257.

If an array of nets of built-in net type is connected to a collapsible concatenation of nets, all nets in the concatenation that are coercible to wreal will be coerced. The type of the coerced wreal nets will be determined by identifying the wreal type that corresponds to the net of built-in net types resolution function described in Table 8-1 on page 257. 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 net type is connected to a non-collapsible concatenation expression no coercion will occur, 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 net type 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 net type is allowed only through the `ie` card mechanism. The following connections are supported with an inserted connect module:

- **Net of built-in net type 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 net type 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 net type 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 net type 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.

Inherited connections on nets of built-in net type are not supported.

User-Defined Net Type and Resolution Function

Like the built-in real SV net type, a user-defined net type is declared with the keyword `nettype`, and includes a data type and optionally a resolution function. For example, the following declares a user-defined net type `wTsum` with a 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 net types are scalar reals and unpacked structs containing real sub elements. Other data types are not supported.

Use the `with` keyword to optionally specify a resolution function to be used to resolve the driven value of nets of user-defined net type, as shown below.

```
nettype T wTsum with Tsum;
```

The resolution function of a net type with a data-type `T` is an SV function with a return type of `T` and 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 the dynamic array input argument.

Calls to only the following system tasks or functions are supported inside the resolution function:

- `$realtime`
- `$display`
- `$size`
- `$realtobits`
- `$bitstoreal`
- `$stime`
- `$time`

Unpacked arrays of scalar real or structure wires of user-defined net type (both with or without user-defined resolution functions) are supported. For arrays of net of user-defined net type, each element of the array is considered as an atomic net.

Note: Declaring a packed, multi-dimensional, or dynamic array of nets of user-defined net type results in an error.

The resolution function of any net of a user-defined net type 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 the net has been defined as the unpacked array 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 net type is set before any `initial` or `always` procedure is started and before the activation of the time 0 resolution call. The default initialization value for a net with a user-defined net type is the default value defined by the data type if no initializer is applied. For a net with a user-defined net type of struct data type, any initialization expressions for the members within the struct are applied.

Assignment to nets of user-defined net type is done with continuous assignments. Continuous assignments can be made with a declaration assignment or with a continuous assignment statement.

A continuous assignment to an atomic net cannot not drive a part of the net; it drives the entire net as per its net type. Therefore, the left-hand side of a continuous assignment to a net of a user-defined net type does not contain any indexing or select operations to the data type of `nettype`. In case of arrays of nets of user-defined net type, 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 delay and net declaration delay are supported on the nets of user-defined net types.

If the internal and external connections to a port are of user-defined net types, they are considered as matching net types and are merged into a single simulated net. If only one of the two connections is of a user-defined net type, then the connections have the matching data types and the port is made `input` or `output` and the connection is treated as a continuous assignment from source to sink.

The `force` and `release` statements are not supported on the user-defined nets. In addition, only the following Tcl commands are supported:

- `describe`
- `value`
- `drivers`
- `deposit`
- `probe -screen`
- `probe -shm`

The following is an example of arrays of net of structure net types without the resolution function:

```
// user-defined data type T
typedef struct {
    real field1;
    real field2;
}T;

// A net type 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
```

The following is an example of arrays of net of structure net types 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 net type 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
```

Using Wreal Concatenation Expressions

The AMS Designer simulator supports the following:

- 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 are the limitations of this feature:

- 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;
  assign d[1] = b;
  ```

  instead of the following:

  ```
  wreal d[0:1];
  wreal a, b;
  assign d = {a, b};
  ```

- Explicit declaration of wreal in non-collapsible concatenation is not allowed and results in an error.

  Note: 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

Creating L2R and R2L Connect Modules

Logic-to-Wreal 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;

endmodule
```

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
```

Note: 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.

Note: The discipline restriction also affects the logic nets. Logic nets in the connect modules cannot inherit from nets with incompatible discipline.

Adding Port Connections through R2L Connect Modules

You can add the following port connections through 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 typedata 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:

Note: 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: insert R2L connect module

      output: insert 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: 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.

Inherited Connections in R2L/L2R Connect Modules

You can create R2L/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 Lout = 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 insertion IE again.

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 can be enabled by using the irun/ncelab command-line option `-amssie`. If a hierarchical signal has both DMS (R2L/L2R) and AMS (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) on page 228).

By default, L2E/E2L IEs use the DRS technology. However, when you specify the `-amssie` option, these IEs use the SIE technology. When a signal in a design requires R-L-E interaction, you should specify the `-amssie` option to enable the elaboration for that design. otherwise, the elaborator will generate an error message.

Using Virtuoso Visualization and Analysis in irun 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 Incisive irun and the Virtuoso ADE flows.

For more information, refer to the *Plotting Wreal Signals* section in the *Virtuoso Visualization and Analysis XL User Guide*.

9 Using irun for AMS Simulation

You can run the Virtuoso® AMS Designer simulator by issuing a single command: `irun`. You can use the `irun` command to specify all your AMS input files and options for simulation, which can be particularly useful in your verification flow.

The `irun` command supports input files from many different simulation and programming languages including Verilog and Verilog-AMS, SystemVerilog, VHDL and VHDL-AMS, SPICE and Spectre. You can use the `irun` command to compile, elaborate, and simulate your mixed-language designs.

To understand how `irun` works, start by reading the "Overview" in the *irun User Guide*. To understand how `irun` works for AMS simulation, refer to the following additional topics:

- irun Command Syntax on page 280
- irun Command-Line Options for AMS on page 281
- Using irun with Spectre and SPICE Input Files on page 302
- Specifying Command-Line Options for Spectre on page 303
- Migrating from Three-Step to irun on page 306
- Examples Using irun for AMS Simulation on page 308
- Creating a Run Script for irun on page 312 in "Using the AMS Designer Simulator for Design Verification" on page 309

See also *irun User Guide* for information on binding rules.

irun Command Syntax

The `irun` command has the following syntax:

```
irun irunOptions sourceFiles
```

For AMS simulation, the `irun` command recognizes the following source file types (*`sourceFiles`*) by their file extensions:

Source File Type	Valid Extensions
Spectre or SPICE	`.scs` or `.sp` **Note:** You can specify additional extensions for analog source files using `-spice_ext`. See also "Using irun with Spectre and SPICE Input Files" on page 302.
Verilog-AMS	`.vams`, `.VAMS` **Note:** You can specify additional extensions for Verilog-AMS source files using `-amsvlog_ext`.
VHDL-AMS	`.vha`, `.VHA`, `.vhams`, `.VHAMS`, `.vhms`, `.VHMS` **Note:** You can specify additional extensions for VHDL-AMS source files using `-amsvhdl_ext`.

See also "Changing the Default Set of File Extensions" in the *irun User Guide* for complete information about the file extensions that `irun` recognizes and how to change the default set. You can also type `irun -helpfileext` to see a list of file extensions.

In the following example, `irun` compiles the `.v` file using `ncvlog`, the `.sv` files using `ncvlog -sv`, and the `.vams` file using `ncvlog -ams`:

```
irun top.sv dut.sv dut2.v dut3.vams
```

After compiling the input files, `irun` automatically runs `ncelab` to elaborate the design and `ncsim` to simulate the design.

Important

You must not specify `-ams` on the `irun` command line unless you want to force `irun` to compile all Verilog and VHDL input files as AMS files.

If you have a design that contains Verilog-A design units, see “Including Structural Verilog-A in a Spectre Netlist” on page 439.

Note: 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” on page 198 for more information.

irun Command-Line Options for AMS

To see the set of command-line options (`irunOptions`) that relate to AMS simulation, type the following on the command line:

```
irun -helpsubject ams
```

Note: You can use the `irun -helpsubject amsspice` to view the `irun` options for AMS-SPICE.

You can specify zero or more of the following AMS-related options (`irunOptions`) on the `irun` command line:

irunOptions	Description
`-ams`	Force `irun` to compile all input files as AMS files (Verilog-AMS, VHDL-AMS) regardless of their file extensions
`-amsconnrules` *`rulesName`*	Specify connect rules to use for automatic connect module insertion (for example, between the default `logic` discipline and the `electrical` discipline) **Note:** You can specify more than one `-amsconnrules` 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. *Tip* If you use the Cadence-installed connect rules, 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. See “ie” on page 120.

irunOptions	Description
`-amsfastspice`	Enable Fast SPICE simulator (UltraSim)

`-amsformat <sst2|psfxl|sst2_all|psfxl_all>`	Controls the storage format for AMS probes.
	
	`sst2`: Sets the Tcl-based probes to use sst2 storage. In this mode, the `rawfmt` option specified in the analog control file is honored. This is the default.
	
	`sst2_all`: Sets the Tcl probes to use sst2 storage. However, this option also overrides the `rawfmt` option specified in the analog control file so that all analog probes are stored in the sst2 format in the SHM database. In other words, this option overrides the `rawfmt` setting in the analog control file to `rawfmt=sst2`.
	
	`psfxl`: Enables the unified PSFXL/SST2 waveform database storage. This mode sets the Tcl analog probes to use psfxl storage in the SHM waveform database. The `rawfmt` option in the analog control file is honored. Note that when `rawfmt=psfxl` or `rawfmt=sst2` is specified, the SPICE probes are stored in the default PSFXL/SST2 database in the `psfxl` format. For all other cases, the `rawfmt` data is stored in the specified format in the `.raw` directory.
	
	`psfxl_all`: Enables the unified PSFXL/SST2 waveform database storage. This mode sets the Tcl probes to use psfxl storage and also overrides the `rawfmt` option specified in the analog control file so that all analog probes are stored in the psfxl format in the SHM database. In other words, this option overrides the `rawfmt` setting in the analog control file to `rawfmt=psfxl`.
	
	To enable `irun` to use the unified PSFXL/SST2 waveform database storage:
	
	1. Specify the location of the default SHM database in the `probe.tcl` file. For example:
	`database -open waves -into resultDirName.shm -default`
	`probe -create -database waves -all -depth all`
	
	2. Specify the `probe.tcl` with the `irun` command as follows:
	`irun -amsformat psfxl_all -input probe.tcl`

irunOptions	Description
`-amsoptie`	Enables the simulator to automatically insert a single bidirectional interface element (IE) in place of multiple IEs. See Hierarchical Interface Element Optimization on page 133 for details.
`-amspartinfo` *`file`*	Mixed-signal partition information
`-amssie`	Enable the new simulation technology, strength-based Interface Element (SIE), which accurately models and simulates the analog/digital interface in a mixed-signal design. See Using the Strength-Based Interface Element (SIE) on page 228 for more information.
`-amsrawdir` *`raw_dir`*	Specify the output raw file directory
`-amsvhdl_ext` *`extension`*	Override extensions for VHDL-AMS source files
`-amsvlog_ext` *`extension`*	Override extensions for Verilog-AMS source files **Note:** 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 `.va` to the list of built-in file extensions for Verilog-AMS source files: `-amsvlog_ext +.va` For more information about changing file extensions, refer to the *Changing the Default Set of File Extensions* section in the *irun User Guide*.
`-analogcontrol` *`file`*	Specify analog simulation control file

irunOptions	Description

`-aps_args`	Specify one or more command-line arguments for running an AMS Designer simulation using the APS solver. You can also include multiple entries of the `-aps_args` parameter on the command line, which are concatenated during command processing.
	
	Note: The `-aps_args` parameter is ignored if the Spectre or UltraSim solver is selected.
	
	Valid APS solver arguments in AMS include:
	
	`+errpreset`
	`+espice`
	`+lmode`
	`*` `+lorder`
	`+lqs`
	`*` `+lqsleep`
	`+lqt`
	`*` `+lqtimeout`
	`+lsusp`
	`+lsuspend`
	`*` `+mt[=`*`N`*`]`
	`*` `+multithread[=N]`
	`*` `+parasitics`

irunOptions	Description

	```
	        +part
	        +query
	*       +rtsf
	        +spice
	        -D
	*       -E
	        -V
	        -W
	        -cmiconfig
	        -cmiversion
	        -espice
	        -h
	        -lsusp
	        -lsuspend
	*       -mt
	*       -multithread
	*       -outdir
	        -plugin plugin_path
	        -proc
	*       -r
	*       -raw
	*         -ahdllint [=value]
	*         -ahdllint_maxwarn=n
	          -ahdllint_log=file
	```
	 See the *Virtuoso Accelerated Parallel Simulator User Guide* for more information about these arguments. **Note:** Arguments marked with * are supported in environment variables `spectre_DEFAULTS` or `SPECTRE_DEFAULTS` in AMS Designer Simulator. If the `spectre_DEFAULTS` (or `SPECTRE_DEFAULTS`) environment variable is specified, the AMS Designer simulator parses the environment variable before the `-aps_args` parameter as the default option. You can achieve parasitics reduction for RF circuits by using the `+parasitics [=`*N* | rf`]` argument with `-aps_args`. `-aps_args +parasitics=[N | rf] ...` Where the value specified for the `+parasitics` argument represents the maximum frequency (in GHz) of interest for RF 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.

irunOptions	Description

	Note: If you specify more than one argument, you must separate them with a space and enclose them within quotation marks.
	
	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.
	
	`irun -aps_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.
	
	`irun -aps_args "+lqs 300"`
	
	The `+lsuspend` argument (applied by default) allows you to suspend or resume the license for APS 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.

irunOptions	Description
	You can use the `-ahdllint` command-line option to turn on the AHDL linter feature that enables you to detect modeling 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 modeling issues. 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 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 forces 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 or Verilog-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 `ncsim` stage.

irunOptions	Description
	Note: 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.
`-chkdigdisp`	Perform digital net's discipline compatibility
`-clean`	Delete 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 `portbind` files, skeletons, and other AMS Designer-specific processing files.
`-cleanlib`	Deletes all `.pak` files found in the libraries specified in the `cds.lib` file that is located in the current working directory or the `cds.lib` file specified using the `-cdslib` option. It also removes the `./INCA_libs` directory. The `-cleanlib` option searches through the entire `cds.lib` structure and also deletes any additional `cds.lib` files that are included in the original. This option does not remove the `.pak` files that do not have write permissions, or the `.pak` files located in the Cadence Install directory. *Caution* ***Use the –cleanlib option carefully because it deletes all the writable .pak files found in the libraries specified in the cds.lib file even if the libraries are shared with another process.***
`-cleanlibverify`	This option behaves similarly as the `-cleanlib` option. However, when this option is used, irun displays the list of files that are being removed and removes the files only after confirmation.
`-cleanlibscript`	This option creates an executable script, which upon executing, produces the same result as the `-cleanlib` option. The script is called `cleanlibscript.sh` and is created in the working directory. After creating the script, irun exits.
`-default_spice_oomr`	

irunOptions	Description
	Assign a default value (`1'bx`) when a digital statement contains an out-of-module reference to a SPICE block See “Using a Command-Line Option to Manage Out-of-Module References to SPICE” on page 188 for more information.
`-discipline` *`disciplineName`*	
	Discipline to use for undisciplined digital wires
`-disres default|detailed|none`	
	Set discipline resolution

irunOptions	Description
-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: (see example below)

```
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 (electrical)
 Port :
testbench.msbuf@ms_buf<module>.I2@ana_nand<module>.I1@my
_inv<module>.in (logic18V input)
 Parameters :
         vsup : 1.8
         vthi : 1.2
         vtlo : 0.6
         tr : 0.0
 List of Ports connected to net testbench.msbuf.I2.Y1 :
(Total: 1)
         testbench.msbuf.I2.I1.in (logic18V input)
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.

Note: The `-iereport` option is aliased to the `-ieinfo` option.

-ieinfo_driverload	Generate a Tcl file, `ieinfo_driverload.tcl`, with the drivers and loads information for both CE and IE.

irunOptions	Description
`-ieinfo_driverload_tcl` <*filename*>	Generate a user-specified Tcl file that contains drivers and loads information for both CE and IE.
`-ieinfo_log` <*filename*>	Specify the file to which the results of the `-iereport` are written. If this option is not specified, the results are written to the default `ams_ieinfo.log` file in the current working directory.
`-ieinfo_probe`	Generate a Tcl file `ieinfo_probe.tcl` that contains the probe-related information for both IE and CE.
`-ieinfo_probe_tcl` <*filename*>	Generate a user-specified Tcl file that contains the probe-related information for both IE and CE.

irunOptions	Description
`-ieinfo_summary`	Generate 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>.\itune[1]
(discipline: electrical, direction: input, nettype:
electrical)

Parameters :

         vsup       :5
         vdelta     :0.078125
         vlo        :0
         vx         :0
         tr         :5e-11
         tf         :5e-11
         ttol_t     :5.000000000000001e-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.

`-ignore_missing_spice_port`

Ignores the missing SPICE ports displayed as "not found" in the port bind file.

irunOptions	Description
`-ignore_spice_oomr`	Ignore any digital statements that contain out-of-module references to SPICE blocks See “Using a Command-Line Option to Manage Out-of-Module References to SPICE” on page 188 for more information.
`-mixed_bus_opt`	Does not allow mixed buses to be automatically generated for unsupported constructs.
`-modelpath` *string*	For Verilog-AMS, specify one or more model files, optionally including a model section specifier such as `-modelpath ./models/resistor.scs(res)`
`-nettype_port_relax`	Allows for relaxed port compatibility rules for connections of built-in net types. See Using Real Number Modeling in SystemVerilog on page 257 for more information.
`-noparamerr`	Do not flag setting undefined parameters as error
`-ppe`	Invoke the post-processing environment (PPE) See Running the SimVision Analysis Environment for more information.
`-propspath` *path*	Specify analog occurrence property database file
`-rnm_package`	Append the `cds_rnm_pkg` package to the `irun` command line. This builds the package and makes it available for import in all SV modules that irun is compiling. See Using Real Number Modeling in SystemVerilog on page 257 for more information.
`-scope_discipline` *disc_scope*	Specify one scope-based discipline
`-solver spectre | ultrasim | aps`	Specify whether the Spectre solver, the UltraSim solver, or the APS solver is to be used with the AMS Designer simulator. If this parameter is not specified with the `irun` command, the Spectre solver is used by default.

irunOptions	Description
-spectre_args	Specify one or more Spectre command-line arguments. You can also include multiple entries of the `-spectre_args` parameter on the command line, which are concatenated during command processing. **Note:** The `-spectre_args` parameter is ignored if APS or UltraSim solver is selected. Valid Spectre arguments in AMS include:

```
*         +aps
          +cktpreset[=value]
          +errpreset
          +espice
          +lmode
*         +lorder
          +lqs
*         +lqsleep
          +lqt
*         +lqtimeout
          +lsusp
          +lsuspend
*         +mt[=N]
*         +multithread[=N]
*         +parasitics
          +query
*         +rtsf
          +spice
          -D
*         -E
          -V
          -W
          -cmiconfig
          -cmiversion
          -espice
          -h
          -lsusp
          -lsuspend
*         -mt
*         -multithread
*         -outdir
          -plugin plugin_path
          -proc
*         -r
*         -raw
*         -ahdllint [=value]
*         -ahdllint_maxwarn=n
          -ahdllint_log=file
```

irunOptions	Description

	Note: Arguments marked with * are supported in environment variables `spectre_DEFAULTS` or `SPECTRE_DEFAULTS` in AMS Designer Simulator. If the `spectre_DEFAULTS` (or `SPECTRE_DEFAULTS`) environment variable is specified, the AMS Designer simulator parses the environment variable before the `-spectre_args` parameter as the default option.
	
	 See the *Virtuoso Spectre Circuit Simulator User Guide* and the *Virtuoso Spectre Circuit Simulator Reference* for information about these arguments. See also "Specifying Command-Line Options for Spectre" on page 303
	
	 You can achieve parasitics reduction for RF circuits by using the `+parasitics [=`*N* | rf`]` argument with the `-spectre_args` parameter.
	
	 `-spectre_args +parasitics=[N | rf] ...`
	
	 Where the value specified for the `+parasitics` argument represents the maximum frequency (in GHz) of interest for RF 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.
	
	 Note: If you specify more than one argument, you must separate them with a space and enclose them within quotation marks like this:
	
	 `irun -spectre_args "-raw ../psf"`
	
	 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.
	
	 `irun -spectre_args "+lqt 0"`

irunOptions	Description
	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. `irun -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. **Note:** Stand-alone Spectre does not have `+lsuspend` by default; this is an AMS-only behavior. This command will have an effect only on the analog licenses. You can use the `-ahdllint` command-line option to turn on the AHDL linter feature that enables you to detect modeling 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 modeling issues. 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 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

irunOptions	Description
	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 forces 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 or Verilog-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. **Note:** The lint checks are performed during `ncsim` stage. **Note:** 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. The `+cktpreset` option, with a possible value of `sampled` (`cktpreset=sampled`) enables Spectre or the APS technologies to take less time steps during conservative mode simulation, while improving simulation resolution at each time step.
`-spectre_e`	Run Spectre parser with the `-E` option
`-spice_ext` *`extension`*	Override extensions for Spectre and SPICE source files See also “Using irun with Spectre and SPICE Input Files” on page 302.

irunOptions	Description
`-top` *`top_unit`*	Specifies *`top_unit`* as the top-level design unit *Important* You must use `-top` if the top level of your design is VHDL. **Note:** `irun` automatically determines the top-level design unit from Verilog or SystemVerilog source files. See `-top` in the *irun User Guide* for more information. See also "Verilog and VHDL-AMS" in the "Compatibility with Existing Use Models" chapter of the *irun User Guide* for information about using `-top` to specify connect modules and cds_globals.

`-ultrasim_args`	Specify one or more UltraSim command-line arguments. You can also include multiple entries of the `-ultrasim_args` parameter on the command line, which are concatenated during command processing.
	
	Note: The `-ultrasim_args` parameter is ignored if the APS or Spectre solver is selected.
	
	Valid UltraSim arguments in AMS include:
	
	`+lorder`
	`+rtsf`
	`-turbo`
	`-plugin` *`plugin_path`*
	
	`*-ahdllint [=`*`value`*`]`
	`*-ahdllint_maxwarn=`*`n`*
	`-ahdllint_log=`*`file`*
	
	Arguments marked with * are supported in environment variables `ultrasim_DEFAULTS` or `ULTRASIM_DEFAULTS` in AMS Designer Simulator.
	
	The `-turbo` argument is used to turn off the UltraSim-Turbo feature, which is available only in `sim_mode=a` and is turned on by default in this mode.
	
	The `+rtsf` argument enables RTSF, which is a PSF extension that can plot extremely large datasets (where signals have a large number of data points, for example 10 million) within seconds.

irunOptions	Description
	You can use the `-ahdllint` command-line option to turn on the AHDL linter feature that enables you to detect modeling 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 modeling issues. Possible values for `-ahdllint` 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 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 forces 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 or Verilog-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. **Note:** 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

irunOptions	Description
	the model, Verilog-A will generate a lint warning but VerilogAMS will not. **Note:** The lint checks are performed during `ncsim` stage. **Note:** If you specify more than one argument, you must separate them with a space and enclose them within quotation marks.
`-uselicense`	Specify colon-separated mnemonics to select license. See "-uselicense Option" on page 459 for more information.
`-rnm_coerce default|none| detailed | off scopeType-scope-`	
	Enables scope-based turning off of wreal coercion. 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. Possible values are: `none` - disable wreal coercion. `detailed` - enable wreal coercion. `default` - enable global coercion with default resolution. `off scopeType` *`scope-`* disable local coercion in scope, coercion in other scope is ON. 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`. Example: `rnm_coerce "off inst-top.dcinst-"` All the net of instance `top.dcinst` and its children will not be coerced to wreal; top level and other instances will be coerced as normal.

irunOptions	Description
`-wreal_resolution` *resolutionFunction*	Specifies the `wreal` resolution function you want the elaborator to use; valid values for *resolutionFunction* are: ■ `default` – Default setting ■ `fourstate` – Verilog 4-state logic resolution algorithm ■ `sum` – Summation of all drivers ■ `avg` – Average of all drivers ■ `min` – Minimum value of all drivers ■ `max` – Maximum value of all drivers See “Selecting a wreal Resolution Function” on page 244 for more information.
`-zparse` *SKILL_file*	Enable zparsing

Note: For information about `irun` options, refer to the *irun User Guide*.

Using irun with Spectre and SPICE Input Files

`irun`'s Spectre and SPICE input file support is consistent with what Cadence's MMSIM technology offers (which applies to the Spectre and UltraSim circuit simulators):

- A `.sp` extension indicates a SPICE syntax input file.
- A `.scs` extension indicates a Spectre syntax input file.

Important

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.

Your top-level analog file can contain SPICE subcircuit definitions, analog models, analog simulation control statements, UltraSim 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. 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
```

Important

If you instantiate SPICE models in your Verilog-AMS code, you must use the `-modelpath` command-line option to declare the analog SPICE model definitions to the digital elaborator. If you instantiate SPICE models only in SPICE code, you do not need to use the `-modelpath` option.

Specifying Command-Line Options for Spectre

You can use the `-spectre_args` command-line option to `irun` (or to `ncsim`) to specify Spectre command-line options such as `+parasitics`. See the following topics for more information:

- Turning On Spectre Multithreading for Device Evaluation on page 303
- Turning Off Spectre Multithreading on page 303
- Turning On Spectre Parasitic Reduction on page 304
- Loading Plug-In for Spectre Netlist Compiled Functions (NCFs) on page 304
- Enabling AMS-APS Mode on page 304

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 mutlithreading for AMS simulation, use the `-spectre_args` command-line option to `irun` (or to `ncsim`) to specify the Spectre `+mt` or `+multithread` command-line argument as follows:

```
irun -spectre_args +mt[=N] ...
irun -spectre_args +multithread[=N] ...
```

where `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 by specifying `-mt` or `-multithread` with the `-spectre_args` command-line option to `irun` (or to `ncsim`) as follows:

```
irun -spectre_args -mt ...
irun -spectre_args -multithread ...
```

Alternatively, you can use the `multithread=off` option setting in a control file to turn off multithreading.

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 turn on Spectre parasitic reduction, use the `-spectre_args` command-line option to `irun` (or to `ncsim`) to specify the Spectre `+parasitics` command-line argument as follows:

```
irun -spectre_args +parasitics[=N | rf] ...
```

where 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.

For information about the Spectre `+parasitic` command-line option, look for “Parasitic Reduction” in the *Virtuoso® Spectre Circuit Simulator User Guide*, version MMSIM 7.0 or later.

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:

```
irun -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 *Virtuoso Spectre Circuit Simulator User Guide* for information about NCFs.

Enabling AMS-APS Mode

To use the APS analog simulator with AMS-Spectre, use the `+aps` argument with the `-spectre_args` option of the `irun` command.

Example:

```
irun -spectre_args "+aps"
```

Migrating from Three-Step to irun

You can run the AMS Designer simulator using `irun` or the three-step method. The following information can help you to migrate from the three-step method to using `irun`.

File/Information for Three-Step	...for irun
`cds.lib` (required)	Unnecessary (remove or convert to `irun` options)
`worklib` (required)	Unnecessary (remove or convert to `irun` options)
`hdl.var` (optional)	Unnecessary (remove or convert to `irun` options)
connect modules and connect rules	Use the `ie` statement in an `amsd` block
model path	Include device model files, optionally with a model section specifier, in a control file along with your `amsd` block For example:

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

File/Information for Three-Step	...for irun
`-propspath` `prop.cfg`	Convert to using an amsd block See information about the AMSCB environment variable in "Using an amsd Block" on page 105; see also "Migrating to an amsd Block from prop.cfg" on page 639.
Tcl probes	No change
analog control file	Specify the analog control file directly on the `irun` command line; you do not need to use the `-analogcontrol` option; include one or more amsd blocks in this file

File/Information for Three-Step	...for irun
discipline	Optional; if you do not specify a discipline (using the `-discipline` command-line option to `irun`), the default discipline is `logic`; see also the `mode` specifier on the `ie` statement
timescale	No change
`-ams` option (required)	Unnecessary (remove); `irun` recognizes AMS input files by their extensions

Here is an example of a three-step example that you can simplify using `irun`:

Three-step example:

```
ncvlog -ams \
        -cdslib cds.lib \
        ./source/digital/*.v

ncelab worklib.testbench \
        connectLib.ConnRules_18V_full \
        -cdslib cds.lib \
        -hdlvar hdl.var \
        -snapshot top:snapshot \
        -amsf \
        -timescale 1ns/100ps \
        -discipline logic \
        -prop prop.cfg \

ncsim top:snapshot \
        -cdslib cds.lib \
        -analogcontrol top.scs \
        -input probe.tcl \
        -simcompat hspice
```

`irun` equivalent:

```
irun  ./source/digital/*.v \
        ./amscf.scs \
        -amsf \
        -timescale 1ns/100ps \
        -input probe.tcl
```

Examples Using irun for AMS Simulation

You can specify different file types on the `irun` command line. `irun` uses the file extension to determine the file type, such as Verilog (`.v`) , Verilog-AMS (`.vams`), and analog (`.scs`). For example, the following `irun` command will run the AMS Designer simulation using the Spectre solver:

```
irun top.v test1.vams test.scs
```

This single `irun` command is equivalent to the following three commands:

```
ncvlog -ams top.v test1.vams
ncelab -snap mysnapshot top
ncsim -ana test.scs mysnapshot
```

See also "Migrating from Three-Step to irun" on page 306.

To specify an AMS Designer simulation using the UltraSim solver, an analog control file called `test.scs`, and an `amsdcb.scs` control file, type the following `irun` command:

```
irun -amsfastspice test.scs amsdcb.scs test.vams
```

This single `irun` command is equivalent to the following three commands (from the three-step approach for compilation, elaboration, and simulation):

```
ncvlog -ams test.vams
ncelab -amsfastspice -snapshot work.top:module -propspath prop.cfg top
ncsim -analogcontrol test.scs work.top:module
```

Note: You could shorten these commands as follows:

```
    ncvlog -ams test.vams
    ncelab -amsf -snap work.top:module -prop prop.cfg top
    ncsim -ana test.scs work.top:module
```

See also "Migrating from Three-Step to irun" on page 306.

To specify an AMS Designer simulation using the APS solver, type the following `irun` command:

```
irun -solver aps top.v test1.vams test.scs
```

This single `irun` command is equivalent to the following three commands:

```
ncvlog -ams top.v test1.vams
ncelab -snap mysnapshot top
ncsim -solver aps -ana test.scs mysnapshot
```

See also "Migrating from Three-Step to irun" on page 306.

10

Using the 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 `irun` to simulate designs containing digital and analog design units. `irun` 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.

See the following topics for more information:

- Creating a Testbench on page 310
- Creating a Run Script for irun on page 312

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.

See the following topics for more information:

- Binding Ports on page 314
- Creating a Customized Port-Bind File on page 319

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
```

   Note: 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
   ```

   See also “Reusing Mixed-Language Testbenches” on page 188.

Creating a Run Script for irun

When creating an `irun` run script for design verification, you need to specify the following items on the `irun` command line:

Item/Option		**Example**
Digital design inputs	`irun`	`./source/digital/*.v \`
AMS control file		`amsdcb.scs \`
Analog solver control file		`top.scs \`
UltraSim solver switch		`-amsfastspice \`
Timescale for undefined Verilog modules		`-timescale 1ns/100ps \`
Tcl probe on behavioral nodes		`-input probe.tcl`

Some things to note:

- You specify the AMS control file on the command line, just like any other input file.
- You specify the analog control file on the command line, just like any other input file.
- You do not need to specify any connect rules (`-amsconnrules`) when you have an `ie` statement in an `amsd` block.

  Note: If you do use a connect rules file, you can specify it as a regular input file, directly on the command line.
- You include your model files 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
```

See also

- Using an amsd Block on page 105
- irun Command Syntax on page 280
- irun Command-Line Options for AMS on page 281
- *irun User Guide*

Creating a Tcl File 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 -default
probe -create -database waves -all -depth all
probe -create -database waves testbench.refclk
probe -create -database waves testbench.clk_p0_1x
probe -create -database waves testbench.clk_p0_4x
probe -create -database waves testbench.p0
```

Note: You can also specify simulation control commands (such as `run`) in a Tcl file.

Creating Analog Probes in the Analog Control File

In addition to specifying analog simulation control statements, you can use the UltraSim `.probe` statement to specify analog probes. For example:

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

*-------------------------------------------------------*
* UltraSim Analysis Options
*-------------------------------------------------------*
.tran 1ns 200ns

*-------------------------------------------------------*
* UltraSim Simulator Options
*-------------------------------------------------------*
*ultrasim: .usim_opt  method=gear2
*ultrasim: .usim_opt progress_p=10

.probe v(*) depth=3 preserve=port
.probe v(testbench.p1.vcom) v(testbench.p1.vcop)

.end
```

The simulator saves all analog waveforms into the same database that contains the digital waveforms (`waves.shm`), so you can display both analog and digital waveforms in the same waveform viewer.

Binding Ports

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
       }
   ```
   Note: The default value for `autobus` on the `portmap` statement is `yes`.
3. Specify the AMS control file directly on the `irun` command line:
   ```
   irun ... 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
```

Note: 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]`. For details about the format of the port-bind file, see "Creating a Customized Port-Bind File" on page 319.

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. See “Creating a Customized Port-Bind File” on page 319 for information about the format of the port-bind file.

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 begin
end
endmodule
```

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. (See Binding Ports by Name on page 317)

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`.

Note: 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`; see “Rules That Apply to Customized Port-Bind Files” on page 321). 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 `portmap` statement to specify the file you saved, and run the simulation again.

Creating a Customized Port-Bind File

The 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 `<>`

Note: These default rules are the same as those that apply when you use `autobus`.

See also:

- Customized Port-Bind File Examples on page 320
- Rules That Apply to Customized Port-Bind Files on page 321

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
+ 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 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 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 `irun`.

- 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 `irun`.

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`.

11

Designing with Multiple Power Supplies

See the following topics for more information:

- Using the ie Statement in an amsd Block for Multiple Power Supply Design on page 324

  You can use this method to 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 on page 327

  You can 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 on page 331

Using the 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.

Note: 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. If you prefer to use these more intricate methods (or perhaps you do not want to use the "full-fast" connect rule), see "Using Block-Based Discipline Resolution for Multiple Power Supply Design" on page 327.

See the following topics for more information:

- Specifying a Supply Value to Apply to a Library, Cell, or Instance on page 324

  Note: For other scopes, see "Scope Assignments" on page 121.
- Customizing Cadence-Installed Connect Rules on page 325
- Locating Cadence-Provided Connect Rules on page 326

Specifying a 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]
      ...
      }
```

Note: For other scopes, see "Scope Assignments" on page 121.

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. (See also "Customizing Cadence-Installed Connect Rules" on page 325.)

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"
     }
```

Important

You must specify the full hierarchical path to the instances.

The software writes information about these connect rules to the `irun.log` file.

Customizing Cadence-Installed Connect Rules

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.

Locating 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.

Using Block-Based Discipline Resolution for Multiple Power Supply Design

The Virtuoso® 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.

Note: For more information about discipline resolution, see "Discipline Resolution Methods" in the *Cadence Verilog-AMS Language Reference*.

See the following topics for information about using block-based discipline resolution for multiple power supply design:

- Preparing Connect Modules, Connect Rules, and Discipline Definitions on page 328
- Specifying Custom Connect Modules, Connect Rules, and Disciplines on the Command Line on page 330

Preparing 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
--------------------------
```

Note: The default discipline is `logic`, which is also of `discrete` domain.

5. Create customized connect rules using your custom disciplines.

   For example, the following connect rules use the `logic18V` discipline:

```
--------------------------------------------------------------------------
// ConnRules Specification
connectrules ConnRules_multpower;
    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_multpower.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.

Note: 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.

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 ncelab/irun command. For example:

```
irun -timescale 1ns/100fs -OVERRIDE_Precision
```

Specifying Custom Connect Modules, Connect Rules, and Disciplines on the Command Line

You can specify your custom connect modules and connect rules on the `irun` command line using the `-amsconnrules` option. For example:

```
irun ConnRules_multpower.vams -amsconnrules ConnRules_multpower ...
```

You can use the `-setdiscipline` option to tell the elaborator which discipline to apply to which design scopes:

```
irun ... -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:

```
irun ... -discipline logic18V ...
```

Creating 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 (CMs). 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.

   Note: 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:

   ```
   irun ... -disres detailed ...
   ```

   Note: You can read more about discipline resolution methods and driver-receiver segregation in the *Cadence Verilog-AMS Language Reference*.

Creating Supply-Sensitive Connect Modules

To create supply-sensitive analog-to-digital and digital-to-analog connect modules (CMs), 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
```

Adding 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 i1 (.y(y), .a(a));

endmodule
```

Note: 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 i1 (.y(y), .a(a));

endmodule
```

For more information, see "Netlisting Inherited Connections" in the *Virtuoso AMS Designer Environment User Guide*.

12 Setting Up for Three-Step Simulation

Note: For information about one-step simulation using `irun`, see "Using irun for AMS Simulation" on page 279.

The `ncvlog` and `ncvhdl` compilers parse and analyze your Verilog, Verilog®-AMS, VHDL, and VHDL-AMS source files, and store compiled objects and other derived data in libraries that follow the Library.Cell:View (L.C:V) approach. While you do not need to provide any setup files to run the NC software (refer to the book *Setting Up Your Environment*), the following configuration files can help you manage your data and control the operation of the software:

File	Description
`cds.lib`	Defines design libraries and associates logical library names with physical library locations.
`hdl.var`	Defines variables that affect the behavior of the software.
`setup.loc`	Specifies the search order to use for finding `cds.lib` and `hdl.var` files.

Note: For more information about the library infrastructure, see the *Cadence Application Infrastructure User Guide*.

See the following topics for more information:

- The Library.Cell:View Approach on page 336
- The cds.lib File on page 337
- The hdl.var File on page 347
- The setup.loc File on page 363
- The Property (prop.cfg) File on page 365
- The Port Mapping File on page 383
- The Verilog File for Port Mapping on page 389

- Using hdl.var and cds.lib to Map Libraries and Views on page 391

The Library.Cell:View Approach

Compiled objects and other derived data are stored in libraries. The library structure is organized according to a Library.Cell:View (L.C:V) approach.

- Library

  A collection of related cells that describe components of a single design (a *design library*) or common components used in many designs (a *reference library*).

  Each library is referenced by a logical name and has a unique physical directory associated with it. You define library names and map them to physical directories in the `cds.lib` file.

  The library used for your current design work is called the *working* or *work* library. You define your current work library by setting a variable in the `hdl.var` file or by using the `-work` command-line option.

- Cell

  A cell is an object with a unique name stored in a library. Each module, macromodule, UDP, entity, architecture, package, package body, connectrules, or configuration is a unique cell.

  The internal intermediate objects necessary to represent a cell are contained in the library database file (.pak file) stored in the library directory.

- View

  A view is a version of a cell. Views can be used to delineate between representations (schematic, VHDL, Verilog-AMS), abstraction levels (behavior, RTL, postsynthesis), status (experimental, released, golden), and so on. For example, you might have one view that is the RTL representation of a particular module and another view that is the behavioral representation, or you might have two different versions of a cell - one with timing and one without timing.

  The internal intermediate objects necessary to represent a view are contained in the library database file (`.pak` file) stored in the library directory.

See also "Using hdl.var and cds.lib to Map Libraries and Views" on page 391 for an example.

The cds.lib File

The `cds.lib` file is an ASCII text file that defines which libraries are accessible and where they are located. The file contains statements that map logical library names to their physical directory paths. During initialization, all software that needs to understand library names read the `cds.lib` file and compute the logical to physical mapping.

You can create a `cds.lib` file with any text editor. The following examples show how library bindings are specified in the `cds.lib` file with the `DEFINE` statement. The logical and physical names can be the same or different.

keyword	logical library name	physical location
`DEFINE`	`lib_std`	`/usr1/libs/std_lib`
`DEFINE`	`worklib`	`../worklib`

You can have more than one `cds.lib` file. For example, you can have a project-wide `cds.lib` file that contains library settings specific to a project (like technology or cell libraries) and a user `cds.lib` file. Use the `INCLUDE` or `SOFTINCLUDE` statements to include a `cds.lib` file within a `cds.lib` file.

Note: If you are doing a pure VHDL or a mixed-language simulation, you must use the `INCLUDE` or `SOFTINCLUDE` statement in the `cds.lib` file to include the default `cds.lib` file located in:

`your_install_dir``/tools/inca/files/cds.lib`

This `cds.lib` file contains a `SOFTINCLUDE` statement to include a file called `cdsvhdl.lib`, which defines the Synopsys IEEE libraries included in the release. If you want to use the IEEE libraries that were shipped with Version 2.1 of the NC VHDL simulator or NC simulator instead of the Synopsys libraries, you must include the `cds.lib` file located in

`your_install_dir``/tools/inca/files/IEEE_pure/cds.lib`

By default, Cadence software searches for the `cds.lib` file in the following locations, defined in the `setup.loc` file. The first `cds.lib` file that is found is used.

- Your current directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- Your home directory

- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site setup, if defined)
- *`your_install_dir`*`/share`

You can edit the `setup.loc` file to add other locations to search or to change the order of precedence to use when searching for the `cds.lib` file. See "The setup.loc File" on page 363.

Each program that reads a `cds.lib` file also has a `-cdslib` option that you can use on the command line to override the search order specified in the `setup.loc` file.

The Work Library

The library used for your current design work is called the *work* library. The work library is the library into which design units are compiled. Like other libraries, the directory path of the work library is defined in the `cds.lib` file.

There are several ways to specify which library is the work library. For Verilog-AMS, you can use compiler directives in the source file, the `-work` command-line option, or variables defined in the `hdl.var` file. See "Controlling the Compilation of Design Units into Library.Cell:View" on page 403 for details. For VHDL, define the `WORK` variable in the `hdl.var` file or use the `-work` option on the command line.

cds.lib Statements

The following list shows the statements you can use in a `cds.lib` file.

Define

Associates the logical library name specified with the *`lib_name`* argument with the physical directory path specified with the *`path`* argument.

Syntax:

```
define lib_name path
```

Examples:

```
DEFINE ttl_lib /usr1/libraries/ttl_lib
DEFINE ttl ./libraries/ttl
```

It is an error to specify the same directory in multiple library definitions.

Undefine

Undefines the specified library. This command is useful for removing any libraries that were defined in other files. No error is generated if *`lib_name`* was not previously defined.

Syntax:

```
undefine lib_name
```

Example:

```
UNDEFINE ttl
```

Include

Reads the specified file as a `cds.lib` file. Use `INCLUDE` to include the library definitions contained in the specified file. An error message is printed if *`file`* is not found or if recursion is detected.

The file to be included does not have to be named `cds.lib`.

Syntax:

```
include file
```

Example:

The following example includes the `cds.lib` file in `/users/$USER`:

```
INCLUDE /users/$USER/cds.lib
```

Softinclude

`SOFTINCLUDE` is the same as the `INCLUDE` statement, except that no error messages are printed if the file does not exist. Using `SOFTINCLUDE` to cause recursion results in an error.

Syntax:

```
softinclude file
```

The following example includes the `cds.lib` file in the `$GOLDEN` directory:

```
SOFTINCLUDE $GOLDEN/cds.lib
```

Assign

Syntax:

```
assign lib attribute path
```

Assigns an attribute to the library. The two supported forms are

`ASSIGN` *`lib`* `TMP` *`directory`*	Specifies a directory to provide temporary storage for a particular previously defined library. See “Binding One Library to Multiple Temporary Storage Directories” on page 343 for details.
`ASSIGN AllLibs TmpRootDir` *`directory`*	Specifies a single directory to provide temporary storage for all the libraries used in a design. Each newly created temporary library has the same name as the corresponding master library.

`TMP` bindings take precedence over `AllLibs` bindings. Bindings you create by specifying the `-CDS_IMPLICIT_TMPDIR` option on the command line take precedence over both `TMP` and `AllLibs` bindings.

For example, your `cds.lib` contains

```
DEFINE amstestLib ./amstest
DEFINE basicLib ./basic
ASSIGN basicLib TMP ./basicTMP
DEFINE analogLib ./analog
```

Consequently, your derived files are written to

```
./amstest
./basicTMP
./analog
```

To your `cds.lib` file, you add the statement

```
ASSIGN AllLibs TmpRootDir ./myTMPs
```

After this addition, your derived files are written to

```
./myTMPs/amstestLib
./basicTMP
./myTMPs/analogLib
```

Unassign

Removes an assigned attribute from the library.

No error is generated if the attribute has not been assigned to the library. If the library has not been defined, an error is generated.

Note: The only supported attributes are `TMP` and `TmpRootDir`.

Syntax:

```
unassign lib attribute
```

Example:

```
UNASSIGN iclib TMP
UNASSIGN AllLibs TmpRootDir
```

cds.lib Syntax Rules

The following rules apply to the `cds.lib` file:

- Only one statement per line is allowed.
- Blank lines are allowed.
- Use the pound sign (#) or the double hyphen (--) to begin a comment. You must precede and follow the comment character with white space, a tab, or a new line.

  Examples:

  ```
  # this is a comment
  -- this is another comment.
  ```
- Keywords are identified as the first non-whitespace string on a line.
- Keywords and attributes are case insensitive.
- You can include symbolic variables (UNIX environment variables like `$HOME` and CSH extensions such as ~ and `~user`).
- Symbolic variables and library path names are in the file system domain and are case sensitive.
- You can enter absolute or relative file paths. Relative paths are relative to the location of the file in which they occur, not to the directory where the software was started.
- Library names and path names reside within the file system name-space. For Verilog-AMS, nonescaped library names are the same as the Verilog-AMS name; for VHDL, nonescaped library names are resolved to lower-case.

  You cannot directly use escaped library names in a `cds.lib` file. To use an escaped name, run the *nmp* program in the *your_install_dir*`/tools/bin` directory to see how escaped library names are mapped to file system names. Then use the mapped name in the `cds.lib` file.

The syntax for the *nmp* program is as follows. Note the trailing space after *`illegal_name`*.

```
nmp mapName {Verilog | NVerilog | Vhdl} Filesys '\illegal_name '
```

For example, to use the library named `Lib*`, you must use the library's escaped name format (`\Lib*`), because "`*`" is an illegal character. To determine the mapped file system name for `\Lib*`, type:

```
nmp mapName Verilog Filesys '\Lib* '
```

The nmp program returns:

```
Lib#2a
```

Use the mapped name (`Lib#2a`) in the `cds.lib` file.

Example cds.lib File

The following example contains many of the statements you can use in a `cds.lib` file. Comments begin with the pound sign (#). See "cds.lib Statements" on page 338 for a description of the `cds.lib` statements.

```
# Assign /usr1/libraries/ic_library to the
# logical library name ic_lib
DEFINE ic_lib /usr1/libraries/ic_library

# Specify a relative path to library aludesign.
# The path is relative to this cds.lib file
DEFINE aludesign ./design

# Read cds.lib from the /users/$USERS directory.
INCLUDE /users/${USERS}/cds.lib

# Read cds.lib from the $CADLIBS directory.
SOFTINCLUDE ${CADLIBS}/cds.lib

# Define a temporary directory and assign the TMP attribute
# to it. The directory ./temp must exist and templib must be
# set to WORK in the hdl.var file in order to compile data
# into it.
DEFINE templib ./temp_lib
ASSIGN templib TMP ./temp
```

Binding One Library to Multiple Temporary Storage Directories

You can bind a previously defined library to a temporary storage directory by using the `TMP` attribute with the `ASSIGN` keyword. This allows multiple designers to reference a common library, but store compiled objects in separate design directories. When the `TMP` attribute is applied to a library, a logical OR operation is performed to include the files present in both the master and TMP directories.

Use the `UNASSIGN` statement to remove the `TMP` attribute before compiling your design units into the master library.

The following steps assign the attribute `TMP` to the library `lsttl` (the environment variable `PROJECT` is set to `/usr1/libs`).

1. Set the environment variable at the command-line prompt.

   ```
   setenv PROJECT /usr1/libs
   ```

2. Set the `cds.lib` file variables.

   ```
   # Define the master library directory
   DEFINE lsttl ${PROJECT}/lsttl_lib
   # Assign a temp storage directory
   ASSIGN lsttl TMP ~/work/lsttl_design
   ```

The attribute `TMP` is now assigned to the library `lsttl`. Subsequent calls to `lsttl` include the contents of both the library (`lsttl`) and the directory (`/usr1/work/lsttl_design`).

The `TMP` attribute can be reassigned to a new value without unassigning it first.

Directory Binding Rules

The following rules apply to binding directories with the `TMP` attribute:

- Only one directory can be bound to a master library using the `TMP` attribute.
- If the referenced library does not exist when the `ASSIGN` command is processed, an error is generated and the command has no effect.

Using Implicit TMP Libraries

Many design environments include shared design libraries with file permissions set to read-only. This protects the library by allowing only authorized users to add, delete, or move design units in the libraries. While elaborating designs that include units from these read-only libraries, the elaborator produces new intermediate files and attempts to store them in the read-only library, something it is not allowed to do. One solution to this problem is to use an explicit TMP library (one created by assigning the `TMP` attribute to a library). However, using explicit TMP libraries requires you to add extra lines to the `cds.lib` file and opens up the possibility that design units could be accidentally recompiled into the TMP library, masking the contents of the shared design library.

To solve this problem, the elaborator can automatically create implicit TMP libraries. If the elaborator needs to produce new intermediate files for a design unit that is in a read-only library that has no explicit TMP library assigned, it automatically creates a TMP library to hold the intermediate files. If you use the `-messages` option, the elaborator generates a message like the following:

```
Using implicit TMP libraries; associated with library worklib
```

These implicit TMP libraries are located in the same directory as the design library that contains the snapshot produced by the elaborator. Each directory for an implicit TMP library is named `inca.`*`library_name`* (for example, `inca.amsLib`, `inca.digLib`).

The AMS Designer simulator searches for intermediate files in the following order:

1. The design library defined in the `cds.lib` file
2. Any existing explicit TMP library associated with the design library

3. An implicit TMP library

Note: The form of implicit TMP library creation described in this section is turned off when you use the `-CDS_IMPLICIT_TMPDir` option on the command line.

Debugging cds.lib Files

You can use the `nchelp -cdslib` command to display information about the contents of `cds.lib` files. This can help you identify errors and any incorrect settings contained within your `cds.lib` files.

Syntax:

```
nchelp -cdslib [cds.lib_file]
```

Examples:

```
nchelp -cdslib
nchelp -cdslib ~/cds.lib
nchelp -cdslib ~/design/cds.lib
```

The following example shows how to display information about the contents of `cds.lib` files. In the example, the `nchelp -cdslib` command displays the contents of the `cds.lib` file that would be used. In this example, the `cds.lib` file is in the current working directory.

```
% nchelp -cdslib

nchelp: v2.1.(b6): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
Parsing -CDSLIB file ./cds.lib.

cds.lib files:
1: ./cds.lib

2: /usr1/lbird/nccoex/tools/inca/files/cds.lib
    included on line 4 of ./cds.lib

3:  /usr1/lbird/nccoex/tools/inca/files/cdsvhdl.lib
    included on line 1 of /usr1/lbird/nccoex/tools/inca/files
    cds.lib

4:  /usr1/lbird/nccoex/tools/inca/files/cdsvlog.lib
    included on line 2 of /usr1/lbird/nccoex/tools/inca/files
    cds.lib

Libraries defined:

Defined in /usr1/lbird/nccoex/tools/inca/files/cdsvhdl.lib:
Line #  Filesys   Verilog  VHDL      Path
------  -------   -------  ----      ----

   1    std       std      STD        /usr1/lbird/nccoex/tools
                                     inca/files/STD

   2    ieee      ieee     IEEE       /usr1/lbird/nccoex/tools
                                     inca/files/IEEE

Defined in ./cds.lib:
Line #  Filesys   Verilog  VHDL      Path
------  -------   -------  ----      ----

   6    alt_max2  alt_max2  ALT_MAX2  ./alt_max2

   7    worklib   worklib  WORKLIB   ./worklib
```

The `cds.lib` file in the working directory includes the `cds.lib` file in `tools/inca/files` under the installation directory. That `cds.lib` file includes two other files.

Here are some common error and warning messages caused by problems with the `cds.lib` file:

```
ncvlog board.v
ncvlog: v2.2.(d5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
ncvlog: *W,DLNOCL: Unable to find a 'cds.lib' file to load in.
ncvlog: *F,WRKBAD: logical library name WORK is bound to a bad
         library name 'worklib'.
```

The `DLNOCL` warning occurs when the software cannot find a `cds.lib` file using the search order specified in the `setup.loc` file.

The `WRKBAD` error occurs when the work library is defined in the `hdl.var` file (for example, `DEFINE WORK worklib`), but the `cds.lib` file does not define the corresponding library (for example, `DEFINE worklib ./worklib`).

```
ncvlog board.v
ncvlog: v2.2.(d5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
```

```
ncvlog: *W,DLCPTH: cds.lib Invalid path '/usr1/clinton/inca/board/worklib'
         on line 7 of ./cds.lib (cds.lib command ignored).
```

The `DLCPTH` warning occurs when the directory path specified in the `cds.lib` file does not exist or is inaccessible. For example, you might have the following line in your `cds.lib` file, but you have not created a `./worklib` physical directory.

```
DEFINE worklib ./worklib
```

The hdl.var File

The `hdl.var` file is an ASCII text file that contains the following settings:

- Configuration variables for your design environment such as:
  - Variables that you can use to specify the work library where the compiler stores compiled objects and other derived data. For Verilog-AMS, you can use the `LIB_MAP` or `WORK` variables. For VHDL, use the `WORK` variable.
  - For Verilog-AMS, variables (`LIB_MAP`, `VIEW_MAP`, `WORK`) that you can use to specify the libraries and views to search when the elaborator resolves instances.
- Variables that allow you to define compiler, elaborator, and simulator command-line options and arguments.
- Variables that specify the locations of support files and invocation scripts.

For information about the syntax rules that apply to the `hdl.var` file, see "hdl.var Syntax Rules" on page 360.

You can have more than one `hdl.var` file. For example, you can have a project `hdl.var` file that contains variable settings used to support all your projects and local `hdl.var` files, located in specific design directories, that contain variable settings specific to each project, such as the setting for the `WORK` variable.

By default, Cadence software searches for the `hdl.var` file in the following locations, defined in the `setup.loc` file. The first `hdl.var` file that is found is used.

- Your current directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- Your home directory
- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site setup, if defined)
- *`your_install_dir`*`/share`

You can edit the `setup.loc` file to add other locations to search or to change the order of precedence to use when searching for the `hdl.var` file. See "The setup.loc File" on page 363.

Each program that reads a `hdl.var` file also has a `-hdlvar` option that you can use on the command line to override the search order specified in the `setup.loc` file.

hdl.var Statements

The following list shows the statements you can use in an `hdl.var` file. *`Variable`* is an alphanumeric variable name. *`Value`* is optional; if provided, it is either scalar or a list. See "hdl.var Variables" on page 351 for a list of `hdl.var` variables.

Note: The variable definitions in the `hdl.var` file are treated as literal strings. Do not use quotation marks in the definitions unless you explicitly want them as part of the input. For example, use:

```
DEFINE NCVLOGOPTS -define foo=16’h03
```

instead of

```
DEFINE NCVLOGOPTS -define foo="16’h03"
```

which is the same as typing:

```
ncvlog -define foo=\"16’h03\"
```

Define

Defines a variable and assigns a value to the variable.

Syntax:

```
define variable value
```

The following example defines the variable `WORK` to be `worklib`.

```
DEFINE WORK worklib
```

The following example defines `VERILOG_SUFFIX` as the list `.v`, `.vg`, and `.vb`.

```
DEFINE VERILOG_SUFFIX (.v, .vg, .vb)
```

The following example defines the variable `NCVHDLOPTS`, which is used to specify command-line options for the `ncvhdl` compiler.

```
DEFINE NCVHDLOPTS -messages -errormax 10
```

Undefine

Causes *`variable`* to become undefined. This statement is useful for removing definitions defined in other files. If *`variable`* was not previously defined, you do not get an error message.

Syntax:

```
undefine variable
```

Example:

```
UNDEFINE NCUSE5X
```

Include

Reads *`filename`* as an `hdl.var` file.

Use `INCLUDE` to include the variable definitions contained in the specified file. The pathname can be absolute or relative. If it is relative, it is relative to the `hdl.var` file in which it is defined.

Syntax:

```
include filename
```

Examples:

```
INCLUDE ~/my_hdl.var
INCLUDE /users/${USER}/hdl.var
```

If the file is not found, a warning message is printed.

Softinclude

`SOFTINCLUDE` is the same as the `INCLUDE` statement, except that no warning message is printed if the specified file cannot be found.

Syntax:

```
SOFTINCLUDE filename
```

Examples:

```
SOFTINCLUDE ~/hdl.var
SOFTINCLUDE ${GOLDEN}/hdl.var
```

Mapn2d

`MAPN2D` declares the specified VHDL-AMS nature to be compatible with the specified Verilog-AMS discipline. You must have a `MAPN2D` statement for every user-defined VHDL-AMS nature that connects to a Verilog-AMS component. When specifying the VHDL-AMS nature, you must use the fully-expanded name of the form *`library.package.nature`*.

Syntax:

```
MAPN2D expanded_VHDL-AMS_nature Verilog-AMS_discipline
```

Example:

```
MAPN2D WORK.electricalSystem.electrical electrical
```

The Virtuoso® AMS Designer simulator provides a default nature-to-discipline mapping for VHDL-AMS natures defined in the IEEE library. You do not need `MAPN2D` statements for these natures. For more information about the default mappings, see "Mapping Verilog-AMS Disciplines to VHDL-AMS Natures" on page 165.

hdl.var Variables

You can use the following variables in an `hdl.var` file:

hdl.var Variables

Modelincdir	Specifies a list of directories to be searched for model files, included files, or files that are passed as instance parameter values.
Modelpath	Specifies SPICE or Spectre source files for the models you use in your design. If a source file is a library file (which begins with the `library` keyword), you must also specify a section for that file. To facilitate use with multi-technology simulation (MTS), the variable also specifies the scope in which the modelpath variable is to be used. (The `modelpath` specification is used in the scope and in scopes below the specified scope.) If the scope specification is omitted, the elaborator uses the global scope by default.
Lib_map (Verilog-AMS only)	Maps files and directories to library names. Use the plus sign (+) to create a default for files or directories that are not explicitly stated.
Ncelabopts	Sets elaborator command-line options. Top-level design unit names can also be included.
Nchelp_dir	Specifies the path to the directory where the help text files are located. These files are used by `nchelp` to print more detailed information on the messages printed by other programs.
Ncsdfcopts	Sets command-line options for `ncsdfc`.
Ncsimopts	Sets simulator command-line options. A snapshot name can also be included.
Ncsimrc	Executes a command file when `ncsim` is started. This command file can contain commands, such as aliases, that you use with every simulation run.
Ncupdateopts	Sets command-line options for ncupdate. For example, if you have compiled a new elaborator with PLI routines statically linked, `ncupdateopts -ncelab` specifies the path to the new elaborator. See Protecting IP Source Files for more information.

hdl.var Variables, *continued*

Ncuse5x	For Verilog-AMS, causes the software to generate three files when you compile Verilog-AMS source files: `master.tag`, `verilog.v`, and `pc.db`. You need these files if you want to view Verilog-AMS objects when you browse libraries using Cadence's Virtuoso® Design Environment software. You also need them if you plan to use a configuration for your design.
Ncvhdlopts (VHDL only)	Sets command-line options for the `ncvhdl` compiler. VHDL source file names can also be included.
NCVLOGOPTS (Verilog-AMS only)	Sets command-line options for the `ncvlog` compiler. Verilog-AMS source file names can also be included.
SRC_ROOT	Defines an ordered list of paths to search for source files when you are updating a design either by running `ncsim -update` or by rerunning `irun`. NC software looks for source files along this ordered list of paths if design units are out of date.
Verilog_suffix (Verilog-AMS only)	Defines valid file extensions for Verilog-AMS source files.
Vhdl_suffix (VHDL only)	Defines valid file extensions for VHDL source files.
View (Verilog-AMS only)	Sets the view name.
View_map (Verilog-AMS only)	Maps files and file extensions to view names.
Work	Defines the current work library into which HDL design units are compiled.

Modelincdir

Specifies a list of directories to be searched for model files, included files, or files that are passed as instance parameter values.

You can also achieve the same result by using the `ncelab -modelincdir` option. If you use both the `MODELINCDIR` variable and the `-modelincdir` option, the latter takes precedence.

For example, you have a file with the following contents.

```
// model.scs
simulator lang=spectre
include "moremodels.scs"
```

```
ahdl_include "mymodel.va"
...
v1 (s gnd) vsource type=pwl file="wave.pwl"
...
```

In addition, the `moremodels.scs` is in the `basemodels` directory and the `mymodel.va` and `wave.pwl` are in the `detailmodels` directory. To ensure that all the models are found, you define an `hdl.var` variable, as follows:

```
DEFINE MODELINCDIR $basedir/basemodels:$basedir/detailmodels
```

Modelpath

Specifies SPICE or Spectre source files for the models you use in your design. If a source file is a library file (which begins with the `library` keyword), you must also specify a section for that file. To facilitate use with multi-technology simulation (MTS), the variable also specifies the scope in which the modelpath variable is to be used. (The `modelpath` specification is used in the scope and in scopes below the specified scope.) If the scope specification is omitted, the elaborator uses the global scope by default.

Note: You cannot use a scope for the `modelpath` variable when you specify a SPICE subcircuit.

Syntax:

DEFINE MODELPATH "[**cell-**[*lib_name*.]*cell_name*[**:***view_name*]**-**] *pathname* [**(***section***)**]{**:** *pathname* [**(***section***)**]}**"**

You may define only one `modelpath` in an `hdl.var` file.

You can achieve the same result by using the `irun -modelpath` option and the `ncelab -modelpath` option. If you use both the `MODELPATH` variable and the `-modelpath` option, the option form takes precedence.

Example: Using the MODELPATH variable

The file `simple_cap.m` contains the following definition which instantiates an analog model, `my_mod_cap`.

```
// simple_cap.m: cap model definition

simulator lang=spectre
parameters base=8

model my_mod_cap capacitor c=2u tc1=1.2e-8 tnom=(17 + base)  w=4u l=4u cjsw=2.4e-10
```

You can use a statement like the following in your `hdl.var` file to use this definition.

```
DEFINE MODELPATH simple_cap.m
```

If you are using sections, use a statement like the following to tell the elaborator to use the `typ` sections for each of the two models:

```
DEFINE MODELPATH mymos(typ):yourmos(typ)
```

Note: You separate the models using a colon, without any spaces.

Example: Mixed SPICE and Spectre Syntax

The following file, which meets the requirements of the Spectre solver, uses only the `simulator lang=spice` statement.

```
section nominal
   simulator lang=spice
   .MODEL MYCAP ...
endsection

section high
   simulator lang=spice
   .MODEL MYCAP ...
endsection
```

If you intend to use this file for the UltraSim solver, you must modify the statements to switch back to Spectre language syntax as follows:

```
simulator lang=spectre
section nominal
   simulator lang=spice
   .MODEL MYCAP ...
   simulator lang=spectre
endsection

section high
   simulator lang=spice
   .MODEL MYCAP ...
   simulator lang=spectre
endsection
```

You can also specify model files for an analog device using the `include` statement in the AMS control file. In this case, you do not require the `-modelpath` option. However, this support is available only in the single-step (`irun`) method and not the three-step method.

Example

```
$ irun work.scs
```

Where `work.scs` contains the following lines:

```
include "a.scs" section="asec" amsd_subckt_bind=yes

include "b.scs" section="bsec" amsd_subckt_bind=yes
```

In multi-technology simulation (MTS) flow, you must change all the MTS lines in the `config` statement of the AMSD block. For example:

```
$ irun work.scs
```

where `work.scs` contains the following lines:

```
amsd {
    config scope = "module_A" model = "modelA.scs" section = "tlib"
    config scope = "module_B" model = "modelB.scs"
    ...
}
```

Lib_map (Verilog-AMS only)

Maps files and directories to library names. Use the plus sign (+) to create a default for files or directories that are not explicitly stated.

For `ncvlog`, `LIB_MAP` specifies that source files are to be compiled into a particular library. See "Controlling the Compilation of Design Units into Library.Cell:View" on page 403.

For `ncelab`, `LIB_MAP` specifies the list of libraries to search when resolving instances. See "Binding during Elaboration" on page 442.

Example:

```
DEFINE LIB_MAP (./source/lib1/... => lib1, \
                ./design => lib2, \
                top.v => lib3, \
                + => worklib )
```

Ncelabopts

Sets elaborator command-line options. Top-level design unit names can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCELABOPTS -messages -errormax 10
```

Nchelp_dir

Specifies the path to the directory where the help text files are located. These files are used by `nchelp` to print more detailed information on the messages printed by other programs.

The help files are usually located in:

`your_install_dir``/tools/inca/files/help`

Use the `NCHELP_DIR` variable if the help files are located in another directory.

```
DEFINE NCHELP_DIR path_to_help_files
```

Ncsdfcopts

Sets command-line options for `ncsdfc`.

Example:

```
DEFINE NCSDFCOPTS -messages -precision 1ps
```

Ncsimopts

Sets simulator command-line options. A snapshot name can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCSIMOPTS -messages -errormax 10
```

Ncsimrc

Executes a command file when `ncsim` is started. This command file can contain commands, such as aliases, that you use with every simulation run.

Example:

```
DEFINE NCSIMRC /usr/design/rcfile
```

Ncupdateopts

Sets command-line options for *ncupdate*. For example, if you have compiled a new elaborator with PLI routines statically linked, `ncupdateopts -ncelab` specifies the path to the new elaborator. See *Protecting IP Source Files* for more information.

Example:

```
DEFINE NCUPDATEOPTS -ncelab ./pli/my_elab
```

Ncuse5x

For Verilog-AMS, causes the software to generate three files when you compile Verilog-AMS source files: `master.tag`, `verilog.v`, and `pc.db`. You need these files if you want to view

Verilog-AMS objects when you browse libraries using Cadence's Virtuoso® Design Environment software. You also need them if you plan to use a configuration for your design.

For VHDL, causes the software to generate a `pc.db` file, which is necessary if you want to use the hierarchy editor.

Example:

```
DEFINE NCUSE5X
```

Ncvhdlopts (VHDL only)

Sets command-line options for the `ncvhdl` compiler. VHDL source file names can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCVHDLOPTS -messages -errormax 10 -file ./proj_file
```

NCVLOGOPTS (Verilog-AMS only)

Sets command-line options for the `ncvlog` compiler. Verilog-AMS source file names can also be included.

You cannot include the `-logfile`, `-append_log`, `-cdslib`, or `-hdlvar` options in the definition of this variable.

Example:

```
DEFINE NCVLOGOPTS -messages -errormax 10 -file ./proj_file
```

SRC_ROOT

Defines an ordered list of paths to search for source files when you are updating a design either by running `ncsim -update` or by rerunning `irun`. NC software looks for source files along this ordered list of paths if design units are out of date.

Search for "SRC_ROOT" in the "Compiling Verilog Source Files" book for more information.

Example:

```
DEFINE SRC_ROOT (~lbird/source, $PROJECT)
```

Verilog_suffix (Verilog-AMS only)

Defines valid file extensions for Verilog-AMS source files.

Example:

```
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)
```

This variable has no effect on the behavior of the Virtuoso® AMS environment.

Vhdl_suffix (VHDL only)

Defines valid file extensions for VHDL source files.

Example:

```
DEFINE VHDL_SUFFIX (.vhd, .vhdl)
```

View (Verilog-AMS only)

Sets the view name.

See “Controlling the Compilation of Design Units into Library.Cell:View” on page 403 for details on this variable.

Example:

```
DEFINE VIEW behavior
```

View_map (Verilog-AMS only)

Maps files and file extensions to view names.

For `ncvlog`, `VIEW_MAP` specifies that files or files with a particular extension are compiled with a specific view name. See “Controlling the Compilation of Design Units into Library.Cell:View” on page 403.

For `ncelab`, `VIEW_MAP` is used to establish the list of views to search when resolving instances. See “Binding during Elaboration” on page 442.

Example:

```
DEFINE VIEW_MAP ( .v => behav, \
                .rtl  => rtl, \
                .gate => gate, \
                myfile.v => gate)
```

Work

Defines the current work library into which HDL design units are compiled.

Example:

```
DEFINE WORK worklib
```

hdl.var Syntax Rules

The following rules apply to the `hdl.var` file:

- Only one statement per line is allowed.
- Keywords and variable names are case insensitive.
- Variable values, file names, and path names are case sensitive.
- Begin comments with either the pound sign (`#`) or a double hyphen (`--`). The comment character must be either the first character in a line or preceded by white space.
- You can extend a statement over more than one line by using the escape character (\ ) as the last character of the line. For example:

  ```
  DEFINE ALPHA (a,\
               b,\
               c)
  ```

  is the same as:

  ```
  DEFINE ALPHA (a, b, c)
  ```

- Left and right parentheses indicate the beginning and end of a list of values.
- Use a comma to separate values in a list.
- You can have a list containing zero elements.

  ```
  DEFINE EMPTY_LIST ( )
  ```

- Any character can be escaped using the backslash (\ ) escape character. Characters should be escaped if the meaning of the character is its ASCII value. For example, the following line defines the variable `JUNK` as `\dump\`.

  ```
  DEFINE JUNK \\dump\\
  ```

  The following example defines `LIST` as `a,b c`.

  ```
  DEFINE LIST (a\,b,c)
  ```

- You can use tilde (~) in *`filename`* or *`value`* to specify:
  - ❑ ~ (or `$HOME`)
  - ❑ ~*`user`* (home of <user>)

  The ~ must be the first non-space character in *`filename`* or *`value`*. For example:

  ```
  DEFINE DIR_RELATION ~/bin != ~lbird/bin
  ```

  expands to:

  ```
  /usr/bin != /usr/lbird/bin
  ```

- The space preceding and following a scalar value is ignored. In the following two lines, the variable `TEST` has the same value (`this is a test`):

  ```
  DEFINE TEST        this is a test
  DEFINE TEST this is a test
  ```

- You can use the dollar sign ($) in *`filename`* or *`value`* to indicate variable substitution. The syntax can be either `$`*`variable`* or `${`*`variable`*`}`, where the left and right braces ({ }) are real characters that mark the beginning and end of the variable name. Variable substitution first searches the `hdl.var` definitions and, if none are found, then searches for environment variables.

  The following example uses the environment variable `$SHELL` to define an `hdl.var` variable:

  ```
  DEFINE MY_SHELL $SHELL
  ```

  In the following example, `LIB_MAP` is defined as:

  ```
  (./source/lib1/... => lib1)
  ```

  Then the variable is redefined as

  ```
  (./source/lib1/... => lib1, ./design => lib2).
  DEFINE LIB_MAP (./source/lib1/... => lib1)
  DEFINE LIB_MAP ($LIB_MAP, ./design => lib2)
  ```

  In the following example, `ALPHA` is defined as `first`. Then `BETA` is defined as `first == one`.

  ```
  DEFINE ALPHA first
  DEFINE BETA ${alpha} == one
  ```

- When a scalar value or file name is specified as a relative path, the path is relative to the location of the `hdl.var` file in which it is defined.

Example hdl.var File

In the following example `hdl.var` file, the `WORK` variable is used to define the work library into which design units are compiled. This library must be defined in the `cds.lib` file. Other variables are defined to list valid file extensions for Verilog-AMS and VHDL source files and to specify command-line options for various programs.

```
# Define the work library
DEFINE WORK worklib

# Define valid Verilog-AMS file extensions
DEFINE VERILOG_SUFFIX (.v, .vr, .vb, .vg)

# Define valid VHDL file extensions
DEFINE VHDL_SUFFIX (.vhd, .vhdl)

# Specify command-line options for the ncvhdl compiler
DEFINE NCVHDLOPTS -messages -errormax 10
```

```
# Specify command-line options for the ncvlog compiler
DEFINE NCVLOGOPTS -messages -errormax 10 -ieee1364

# Specify command-line options for the elaborator
DEFINE NCELABOPTS -messages -errormax 10 -ieee1364 -plinooptwarn

# Specify the simulation startup command file
DEFINE NCSIMRC /usr/design/simrc.cmd
```

Debugging hdl.var Files

You can use the `nchelp -hdlvar` command to display information about the contents of `hdl.var` files. This can help you identify incorrect settings that may be contained within your `hdl.var` files.

Syntax:

nchelp -hdlvar [*hdl.var_file*]

Examples:

```
nchelp -hdlvar

nchelp -hdlvar ~/hdl.var
```

The following example shows how to display information about the contents of an `hdl.var` file. In the example, the `nchelp -hdlvar` command displays the contents of the first `hdl.var` file found using the search order in the `setup.loc` file. In this example the `hdl.var` file is in the current working directory.

```
nchelp -hdlvar
nchelp: v2.2.(d5): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
Parsing -HDLVAR file ./hdl.var.
hdl.var files:
1:  ./hdl.var
Variables defined:
Defined in ./hdl.var:
Line #  Name            Value
------  ----            -----
   5    LIB_MAP         ( /net/foghorn/usr1/belanger/chip1 => chip1 , \
                          /net/foghorn/usr1/belanger/libs/misc.v => misc )
   1    NCVLOGOPTS      -messages
   2    NCELABOPTS      -messages
   3    NCSIMOPTS       -messages
   4    VERILOG_SUFFIX  ( .v , .vlog )
   6    VIEW_MAP        ( .g => gates , .b => behav , .rtl => rtl )
   7    WORK            worklib
```

The setup.loc File

By default, programs and utilities that need to read library definition files (`cds.lib`) and configuration files (`hdl.var`) search for these files in the following locations:

- Your current directory
- `$CDS_WORKAREA` (user work area, if defined)
- `$CDS_SEARCHDIR` (if defined)
- Your home directory
- `$CDS_PROJECT` (project area, if defined)
- `$CDS_SITE` (site setup, if defined)
- *`your_install_dir`*`/share`

This search order is defined in a `setup.loc` file located in *`your_install_dir`*`/share/cdssetup/setup.loc`.

To change the default search order or to add new locations to search:

1. Define the `CDS_SITE` environment variable. For example,

   ```
   setenv CDS_SITE your_install_dir/share/local
   ```

2. Copy *`your_install_dir`*`/share/cdssetup/setup.loc` and edit the file to list the directories in the order you want them searched.

The following rules apply to creating a `setup.loc` file:

- Only one entry per line.
- Use a semi-colon (`;`), the pound sign (`#`), or a double hyphen (`--`) to begin a comment.
- The file can include:
  - `~`
  - `~user`
  - `$`*`environment_variable`*
  - `${`*`environment_variable`*`}`

    By convention, environment variables are given uppercase names. See the documentation for your implementation of UNIX for complete details on setting environment variables.

  If a directory specified in `setup.loc` references an environment variable that is not set, the location referenced by the variable is not searched and no warning or error message is issued.

- Relative paths in `setup.loc` files are relative to the current directory; they are not relative to the location of the file in which they occur or to the directory where you started the software.

If a directory specified in `setup.loc` cannot be found or is not accessible, the search advances to succeeding locations without printing warning or error messages.

The Property (prop.cfg) File

A property file is an ASCII file that specifies properties either globally or for specified paths, instances, and cells. You can use this file to specify any properties that can appear in the netlist, including predefined properties and other properties that you define.

When you use the Cadence hierarchy editor to change properties, AMS Designer automatically associates a single property file with the current configuration by placing the property file in the directory that contains the `expand.cfg` file for the configuration. When used like this, the property file is always named `prop.cfg`. The hierarchy editor does not read property files with any other name.

You can also use a text editing program to create a property file that is not associated with a configuration, and in this case the property file can have any name you choose. You use the `irun -propspath` option or the `ncelab -propspath` option to pass a property file to the software. For example, to direct the elaborator to use a property file named `myprops.cfg`, you type a command such as the following:

```
ncelab -messages -propspath myprops.cfg top
```

See the following topics for more information:

- Property File Syntax on page 365
- Property File Precedence on page 368
- Predefined Properties on page 369

Property File Syntax

The property file consists of statements generated according to the following syntax.

```
property_file_statement ::=
        file-format-id integer.integer ;
    |   comment
    |   default property_spec | { property_spec { property_spec } }
    |   cell cellview property_spec | { property_spec { property_spec } }
    |   inst instance property_spec | { property_spec { property_spec } }
    |   path path_spec property_spec | { property_spec { property_spec } }

comment ::=
        one_line_comment
    |   multiple_line_comment

property_spec ::=
        [non-inherited] [no-eval] prop_type prop
            property = prop_value ;
        |   { property = prop_value ; { property = prop_value ; } }

instance ::=
        [( cellview ).]instance_name
```

```
path_spec ::=
        ( cellview ){.[(cellview)]instance_name}.instance_name
cellview ::=
        [lib.]cell[:view]
prop_type ::=
        int
    |   double
    |   string
```

Keyword or Syntax Item	Description
`file-format-id`	Keyword indicating an identifier for the property file of the form *`integer.integer`*. Cadence software uses this keyword and identifier; you do not need to use this keyword and identifier in your custom property files.

`default`	Keyword indicating that the following `property_specification` specifies default properties for all design objects in the design. For example,
	`default{`
	`   string prop sourcefile_opts="-auto_bus`
	`   -bus_delim _";`
	`}`
	applies the specified `sourcefile_opts` values to every cell, inst, or path-based sourcefile binding specified later in the `prop.cfg` file. You can override default properties specified in this way using properties specified with the `cell`, `inst`, or `path` keywords.

`cell`	Keyword indicating that all instances of the specified cell are bound to the same property.
`inst`	Keyword indicating that all instances with the specified name are bound to the property.
`path`	Keyword indicating that the specified instance on the specified path is bound to the property.
`lib`	Library containing the objects to which the property applies. If *`lib`* is not specified, all libraries are considered.
`cell`	Cell to which the property applies.
`view`	Cellview to which the property applies. If *`view`* is not specified, all views are considered.

Keyword or Syntax Item	Description
`instance_name`	Instance or instances to which the property applies. Within a `path` statement, *`instance_name`* can be an asterisk (`*`), which matches everything in that level of the path. An *`instance_name`* that follows a period is an *instance that is instantiated inside* the previous cellview. An *`instance_name`* that follows a cellview without an intervening period, is an *instance of* the previous cellview.
`one_line_comment`	Comment that begins with a double slash (`//`) and continues to the end of the line.
`multiple_line_comment`	Comment that begins with a slash asterisk (`/*`) and continues to the next asterisk slash (`*/`), whether the `*/` marker is on the same line or a different line.
`non-inherited`	Keyword indicating that the property is not to be inherited by instances lower in the hierarchy.
`no-eval`	Keyword indicating that the environment variables found in the property value are not to be evaluated.
`property`	Name of the property to be applied. Property names must begin with `A-Z`, `a-z`, or underscore (_). Subsequent characters can be any of these, `0-9`, or hyphen (`-`). No quotes or white spaces are allowed. Property names are case sensitive.
`prop_value`	The value to be assigned to *`property`*. The value can consist of any of the characters that can be used for the property name or can be a string, in quotation marks (`"`), of printable characters. To include a quotation mark in the quoted string, use `\"`. To include a backslash, use `\\`. The format of the value must match the data type specified by the `int`, `double`, or `string` keyword used in the statement.
`int`	Keyword indicating that *`property`* is an integer.
`double`	Keyword indicating that *`property`* is a double.

Keyword or Syntax Item	Description
`string`	Keyword indicating that *`property`* is a string. When you use this `prop_type`, the *`prop_value`* must be given in quotation marks.

Property File Precedence

With several different ways of specifying property values, it is possible to specify different values for the same property on the same instance. To resolve these conflicts, AMS Designer uses the following rules.

- The statements within a property file are of different precedences. From highest to lowest precedence, they are
  - `path` statement
  - `inst` statement
  - `cell` statement
  - `default` statement

  For example, using the statements

  ```
  default int prop fingers = 7;
  inst i3 int prop fingers = 5;
  ```

  in a property file, sets the value of the `fingers` property to `5` for every instance named `i3`.

- Within any one of the first three statement types, conflicting property values resolve in favor of the statement that uses the most precise path specification.

  For example, using the statements

  ```
  cell lib1.cell2 int prop fingers = 4;
  cell cell2 int prop fingers =3;
  ```

  in a property file, sets the value of the `lib1.cell2 fingers` property to `4`.

- Within a hierarchical level, conflicting property values resolve in favor of the statement that is most specific, so that the precedence of instances, from highest to lowest, is
  - **`(`***`lib`***`.`***`cell`***`:`***`view`***`).`***`inst`*
  - **`(`***`cell`***`:`***`view`***`).`***`inst`*
  - **`(`***`lib`***`.`***`cell`***`).`***`inst`*

- ❑ `(`*`cell`*`).`*`inst`*
- ❑ *`inst`*
- ❑ `(`*`lib`*`.`*`cell`*`:`*`view`*`).*`
- ❑ `(`*`cell`*`:`*`view`*`).*`
- ❑ `(`*`lib`*`.`*`cell`*`).*`
- ❑ `(`*`cell`*`).*`
- ❑ `*`

  For example, using the statements

```
inst (mycell:myview).I3 int prop fingers = 5;
inst (mylib.mycell).I3 int prop fingers = 4;
```

  in a property file, sets the value of the `(mylib.mycell:myview).I3` instance [and of `(mycell:myview).I3` instances from any other library] to 5.

- In multiple `path` statements, conflicting property values resolve in favor of the statement with the path that is the most specific at the right end.

  For example, using the statements

```
path (mylib.mycell:myview).(mycell2)i1.i2 int prop fingers = 5;
path (mycell).(mylib2.mycell2:myview2)i1.i2 int prop fingers = 4;
```

  in a property file, sets the value of the `(mycell).(mylib2.mycell2:myview2)i1.i2` instance to 4.

Predefined Properties

Cadence provides several predefined properties, including the following.

- hdl_cell Property on page 370
- sim_mode Property on page 372
- sim_stub Property on page 372
- sourcefile Property on page 374
- sourcefile_opts Property on page 375
- speed Property on page 382
- verilogfile Property on page 382

hdl_cell Property

The `hdl_cell` property works together with the `sourcefile` property to specify that a Verilog-AMS block instantiates a SPICE block that, in turn, instantiates a Verilog-AMS block. This arrangement is called *SPICE-in-the-middle*. You use the `sourcefile` property to specify the SPICE file and the `hdl_cell` property to specify the Verilog-AMS module that is instantiated in the SPICE block.

You can use the `hdl_cell` property only in the section of the `prop.cfg` file defined by the `default` keyword.

The *`prop_value`* for the `hdl_cell` property is the following. Because the `hdl_cell` property is a string `prop_type`, the *`prop_value`* must be enclosed in quotation marks.

`hdl_cell {hdl_cell}`	Specifies that the Verilog-AMS module *`hdl_cell`* is instantiated in one or more SPICE blocks. If multiple Verilog-AMS blocks are instantiated in SPICE blocks, you can specify multiple *`hdl_cell`* names. All Verilog-AMS modules that are called from SPICE blocks must be declared by an `hdl_cell` property in the `default` section of the prop.cfg file. For example, `string prop hdl_cell="a0 a1 a2"` specifies that blocks `a0`, `a1`, and `a2` are instantiated in one or more SPICE blocks.

Restrictions

- You can only use the `hdl_cell` property (and therefore SPICE-in-the-middle) when you run the AMS Designer simulator with the UltraSim solver in the three-step command-line flow and with the Spectre solver when you use the simulation front end (SFE) parser. You cannot use the `hdl_cell` property when you use any of the following:
  - AMS Designer simulator running in the AMS environment
  - AMS Designer simulator running in the analog design environment (ADE)
  - `irun` command
- You cannot have more than one level of SPICE-in-the-middle. For example, you cannot have Verilog-AMS, SPICE, Verilog-AMS, SPICE, Verilog-AMS.

- You cannot have SPICE-in-the-middle when you are using 5x configurations.

Example

You have a Verilog-AMS module named `verilog_top` that instantiates a SPICE subcircuit named `spice_middle`. The `spice_middle` subcircuit instantiates another Verilog-AMS module named `verilog_bottom`.

The file `verilog_top.v` contains

```
module verilog_top;
   wire [0:5] v;
   ...
   spice_middle xspice_middle ( .p (v) );
   ...
endmodule
```

The file `spice_middle.cir` contains

```
.subckt spice_middle p<0> p<1> p<2> p<3> p<4> p<5>
   ...
xverilog_bottom p<0> p<1> verilog_bottom
   ...
.ends
```

To specify that there are blocks using SPICE-in-the-middle, you include the `sourcefile` and `hdl_cell` properties in the `prop.cfg` file, like this.

```
default
   {
      string prop hdl_cell="verilog_bottom";
   }
cell spice_middle
   {
      string prop sourcefile="spice_middle.cir";
      string prop sourcefile_opts="-auto_bus -bus_delim <>";
   }
```

The `hdl_cell` property (in the `default` section, as required) specifies that the `verilog_bottom` Verilog-AMS module is instantiated in one or more SPICE blocks (but does not indicate which SPICE blocks).

The `sourcefile` property specifies the name of the SPICE file that instantiates `verilog_bottom`. The `sourcefile_opts` property with the `-auto_bus` and `-bus_delim` *`prop_values`* specifies automatic bus binding between SPICE and Verilog-AMS blocks and specifies the bus delimiters that are used.

sim_mode Property

The `sim_mode` property controls the simulation mode that the UltraSim solver uses for the circuit. (The Spectre solver ignores the `sim_mode` property setting.) Usually, you set the `sim_mode` property for cells or instances. For information about the supported modes, see “Simulation Modes” in the *Virtuoso UltraSim Simulator User Guide*.

When designs include VHDL-AMS behavioral devices, the software sets `sim_mode=a` internally. You can override this internally set value by specifying a different `sim_mode` in the simulation control file, but be aware that other values sometimes result in convergence problems or simulation failures.

sim_stub Property

The `sim_stub` property removes a bound schematic, Verilog-A, Verilog-AMS, Spectre, or SPICE block from the configuration and replaces the schematic or block with a stub module that is empty except for interfaces and discipline declarations. By eliminating portions of your design, this property helps you identify blocks that slow down simulation. Be aware, however, that your design might not simulate correctly if you eliminate the circuitry for essential functions such as signal generation.

You can specify the `sim_stub` property with the `cell`, `inst`, and `path` keywords, but not with `default`. You cannot use the `path` keyword when you use the `irun` command.

You cannot use the `sim_stub` property for a module that you also specify using the `verilogfile` property.

Important

If you are migrating from the three-step approach (`ncvlog`/`ncelab`/`ncsim`) to the one-step approach (`irun`), you must change the `sim_stub` property from `"`*`lib.cell`*`:module"` to `"`*`lib.cell`*`:vams"`.

The *`prop_value`* for the `sim_stub` property can be any of the following. Because the `sim_stub` property is a string `prop_type`, you must enclose the *`prop_value`* in quotation marks.

`lib``.`*`cell`*`:`*`view`*	Used to stub out Verilog-AMS modules. Creates the stub module from the specified Verilog-AMS cellview.

{**-cell** *cell*} **-src** *sourcefile*	Used to stub out SPICE or Spectre blocks. Creates the stub module from the specified sourcefile (which is expected to be bound by the `sourcefile` property). The sourcefile must be a SPICE or Spectre file. The `sim_stub` property is not supported for Verilog (digital) or for Verilog-A files bound by the `sourcefile` property.
Null (specified by **""**)	Used to stub out SPICE or Spectre blocks. SPICE and Spectre blocks can be stubbed out with a null specification when you use cell-, inst-, or path-based properties. When you use a null specification to stub out SPICE or Spectre blocks ■ The source file must be specified with the `sourcefile` property. ■ The cell must be specified with the `sourcefile_opts -subckt` property. Using the `sim_stub -cell` and `-src` property values has a higher precedence than using the `sourcefile` and `sourcefile_opts` properties.

You can also stub out SPICE blocks by specifying the `sourcefile` and `sourcefile_opts` properties. The `sourcefile` property is equivalent to `sim_stub -src` and the `sourcefile_opts -subckt` property value is equivalent to `sim_stub -cell`. For example, the following two property specifications are equivalent.

```
inst I22
    {
        string prop sim_stub="-cell top -src /home/jdoe/spicefile.sp";
    }
```

is equivalent to

```
inst I22
    {
        string prop sourcefile="/home/jdoe/spicefile.sp";
        string prop sourcefile_opts="-subckt top";
        string prop sim_stub="";
    }
```

Examples

To stub out instance `I22` whose Verilog-AMS master cell is `foo` in an instance-based binding configuration, you define a property as follows.

```
inst I22
    {
        string prop sim_stub="worklib.foo:module";
    }
```

To stub out instance `I22` whose SPICE master subcircuit is `foo` in an instance-based binding configuration, you define a property as follows.

```
inst I22
    {
        string prop sim_stub="-cell foo -src /home/jdoe/spicefile.sp";
    }
```

sourcefile Property

The `sourcefile` property tells AMS Designer where to find the source files for HSPICE, SPICE, Spectre, or Verilog-A design blocks that describe such components as amplifiers, PLLs, and memories, or where to find the source files for device models wrapped in subcircuits, so that you can use these design blocks in hierarchy editor configuration files without having to supply a model via the model path. You specify the `sourcefile` property as follows:

```
string prop sourcefile="pathToSourceFile";
```

For example:

```
string prop sourcefile="/home/jdoe/spicefile.sp";
```

Note: You must enclose the *`pathToSourceFile`* in quotation marks because `sourcefile` is a string property.

You do not need to use the `sourcefile` property for models or primitives such as transistors.

You can also specify SPICE or Spectre files for AMS Designer simulation using the `MODELPATH` variable in `hdl.var` or the `ncelab -modelpath` option. If you use one of these methods together with the `sourcefile` property, you must not define the same object both with the `sourcefile` property and by specifying a modelpath. If you do, you will get a redefinition error.

For information about specifying how to connect Verilog vector buses to SPICE buses, see "sourcefile_opts Property" on page 375.

sourcefile_opts Property

If you are using the AMS Designer simulator with the Spectre solver and the simulation front end (SFE) parser or with the UltraSim solver, you can use the `sourcefile_opts` property together with the `sourcefile` property to specify how to connect Verilog-AMS (including Verilog) vector buses to SPICE buses. The `sourcefile` property specifies SPICE subcircuits that you instantiate in Verilog-AMS modules; the `sourcefile_opts` property specifies how to handle the involved ports. You can specify port mapping rules in a file using the `-portmap_file` property value (see "The Port Mapping File" on page 383 for more information) as well as specifying options explicitly using the other `sourcefile_opts` property values. Options you specify in a port mapping file override those you specify explicitly using the other `sourcefile_opts` property values.

Note: When you use `sourcefile` together with `sourcefile_opts`, you are set up to use the AMS Designer simulator in your verification flow.

The *`prop_value`* for the `sourcefile_opts` property can be any of the following and you must enclose the *`prop_value`* in quotation marks because `sourcefile_opts` is a string property (see also "Restrictions" on page 381 and "Examples" on page 381):

`-auto_bus` | **`-no_bus`**	If you do not use the `-portmap_file` property value, you must specify either `-auto_bus` or `-no_bus` using the `sourcefile_opts` property. `-auto_bus` specifies that you want Verilog-AMS vector buses automatically mapped and connected to SPICE ports. `-no_bus` specifies that you want Verilog-AMS and SPICE buses not to be connected. Default Value: None.

`-bus_delim empty` | *`delimiters`*

Specifies bus delimiters. `-bus_delim empty` indicates that there are no bus delimiters.

If you use multiple bus delimiters, you can specify the `-bus_delim` *`prop_value`* as often as necessary.

For example, if the bus bits are:

```
a0, a1, a2, a3
```

you might specify

```
string prop sourcefile_opts="-bus_delim empty";
```

If the bus bits are:

```
a<0>, a<1>, a<2>, b_1, b_2, b_3, c[1], c[2], c[3]
```

you might specify

```
string prop sourcefile_opts="-bus_delim <> -bus_delim _
                                -bus_delim []";
```

Valid Values:

Binary delimiters:	`[] <>`
Unary delimiters:	`_ ! @ # $ % ^ & *`
Null delimiter:	`empty`

Default Value:
The elaborator treats “`[]`” and “`<>`” as bus delimiters except when you specify the `-bus_delim` *`prop_value`* explicitly.

`-case_map keep | lower | upper`

Specifies the handling for case mismatches when mapping between Verilog-AMS instantiations and SPICE subcircuit names. The `keep` value means letters are left unchanged; `lower` changes letters to lower case; `upper` changes letters to upper case.

For example, with this module

```
module verilog_r;
   SUB1 XSUB1 (A, B);
...
```

and this SPICE subcircuit

```
.SUBCKT A B SUB1
   ...
.ENDS
```

you can override the default lowercase mapping for SPICE subcircuits for cell `SUB1` in your `prop.cfg` file as follows:

```
cell sub1
{
   string prop sourcefile_opts= "-case_map upper"
}
```

The case of the subcircuit master name you use in an instantiation must match the binding you specify using `-case_map`. For example, if you use `-case_map lower`, you must instantiate SPICE subcircuit `ANALOG_CELL`, defined as

```
.SUBCKT ANALOG_CELL VIN<1> VIN<0> VOUT<1> VOUT<0>…
```

in a Verilog-AMS module as

```
analog_cell IO(.vin(vin), .vout(vout));
```

rather than as

```
ANALOG_CELL IO(.VIN(vin), .VOUT(vout));
```

Default Value: `keep`

-exclude_bus *delim_prefix*

Specifies ports to be excluded from consideration as buses.

For example, for bus bits `bus0`, `bus1`, `bus2` and scalar nets `net0`, `net1`, `net2`, using only `-bus_delim empty` causes the elaborator to treat `net0`, `net1`, and `net2` as buses, which is incorrect. To exclude `net*` as buses, you use

```
string prop sourcefile_opts=
    "-bus_delim empty
     -exclude_bus net";
```

For bus bits `a_1`, `a_2`, `a_3` and scalar nets `out_a`, `out_b`, `in_a`, `in_b`, using the following property prevents treating the scalar nets `out_*` and `in_*` as bus bits:

```
string prop sourcefile_opts="-bus_delim _
     -exclude_bus out -exclude_bus in";
```

-in_port | -out_port *port_basename*

Specifies the direction of a subcircuit port so that you can control the direction of interface elements that involve the port. (By default, the elaborator treats all subcircuit ports as bidirectional.) Specifying the port direction also allows you to maintain the same signal flow direction as the original design, which can make it easier to debug problems that might arise.

port_basename refers to the part of a port name before the delimiter. For example, ports `in_1`, `in_2`, and `in_3` use underscore as a delimiter, so the *port_basename* is `in`. Ports `out1` and `out2` use the empty delimiter, so the *port_basename* is `out`.

You can specify only a single *port_basename* with each `-in_port` or `-out_port` *prop_value*, but you can specify more than one `-in_port` and `-out_port` *prop_values*, if necessary.

For example, if you have input ports `in1`, `in2`, `in3`, `ain1`, `ain2` and output ports `pout1`, `pout2`, `cout1`, and `cout2`, the following property specifies that ports `in*` and `ain*` are input ports and ports `pout*` and `cout*` are output ports:

```
string prop sourcefile_opts="-bus_delim empty
     -in_port in -in_port ain -out_port pout
     -out_port cout"
```

Any unspecified ports are considered `inout` ports.

Restriction: You cannot use `-in_port` and `-out_port` property values if you use the `default` keyword to specify the property.

-no_bus See `-auto_bus`.

-portmap_file *subckt_name*.**pb**

Specifies the name of the port mapping file for a SPICE subcircuit that defines how to connect Verilog and SPICE.

Note: Settings in the port mapping file override other `sourcefile_opts` settings. See "The Port Mapping File" on page 383 for more information.

-subckt *subckt_name*

Specifies the master name of a SPICE subcircuit you want to instantiate in Verilog-AMS. You must use this property value for `inst` or `path` bindings, and *only* for `inst` or `path` bindings.

For example, you instantiate instance `I1` of subcircuit `sub1` in Verilog-AMS module `foo` as follows:

```
.subckt A B sub1
...
module foo
sub1 I1(a, b)
...
```

To specify the master name of the subcircuit in an instance-based property, type the following string property in your property file:

```
inst I1
{
string prop sourcefile_opts="-auto_bus -subckt sub1";
}
```

As a second example, the following property specification indicates that the master name of the subcircuit bound to `top.x1.xinv` is `sub1`:

```
path top.x1.xinv
   {
      string prop sourcefile="analog_top.cir";
      string prop sourcefile_opts="-auto_bus
         -bus_delim <> -subckt sub1";
   }
```

Restriction: You cannot use the `-subckt` property value if you use the `default` keyword to specify the property.

Default Value: None.

-veri_file *file_name*

Specifies a file containing a Verilog module that defines the port mappings to use from a Verilog parent to a SPICE subcircuit instance. See “The Verilog File for Port Mapping” on page 389 for more information.

Restrictions

The following restrictions apply to using the `sourcefile_opts` property (and to using the AMS Designer simulator in your verification flow):

- You cannot use the `irun` command when you specify the property with the `path` keyword.
- You cannot have reverse bus connections.

  For example, the elaborator issues an error for the following Verilog-AMS to SPICE bus connection, even when you specify `sourcefile` and `sourcefile_opts` properties properly.

  ```
  module verilog;
     wire [0:5] v;
     analog_top xana_top ( .p (v) );
  endmodule
  ```

  ```
  .subckt analog_top p<5> p<4> p<3> p<2> p<1> p<0>
  ...
  .ends
  ```

Examples

For example, you have a Verilog-AMS module named `verilog` that calls a SPICE subcircuit named `analog_top` with vector bus bindings.

The file `verilog.v` contains

```
module verilog;
   wire [0:5] v;
   analog_top xana_top ( .p (v) );
endmodule
```

The file `analog_top.cir` contains

```
.subckt analog_top p<0> p<1> p<2> p<3> p<4> p<5>
   ...
.ends
```

In the `prop.cfg` file, you specify the `sourcefile_opts` property as

```
cell analog_top
   {
      string prop sourcefile="analog_top.cir";
      string prop sourcefile_opts="-bus_delim <> ";
   }
```

The `sourcefile` property specifies the name of the SPICE file. The `sourcefile_opts` property specifies Verilog-AMS to SPICE bindings and also specifies the bus delimiters.

See also the `prop.cfg` file in your Cadence installation hierarchy at

```
$AMSHOME/tools/amsd/samples/busConn_stubView
```

for another example illustrating Verilog-AMS to SPICE bus connections.

speed Property

The `speed` property establishes the trade-off between simulation performance and accuracy by adjusting the simulation tolerance. Usually, the `speed` property is set for cells or instances. The `speed` property is used by the UltraSim solver but the Spectre solver ignores the property. For information about the supported values, see the discussion of the `speed` option in "Immediate Set Options (options)" on page 58.

verilogfile Property

The `verilogfile` property specifies Verilog-A, Verilog-AMS, or Verilog (digital) text modules to be brought into libraries so that the modules are available for use in design configurations. Using a `verilogfile` property causes the AMS Designer simulator to use the module definition in the referenced file when the cell, instance, or occurrence is simulated.

The modules bound by using the `verilogfile` property are compiled automatically when you use the AMS environment, the ADE environment, or the `amsdesigner` command. However, when you run the AMS Designer simulator using commands such as `ncvlog`, `ncelab`, and `ncsim`, you must compile these bound modules manually.

Because the `verilogfile` property is a string `prop_type`, the *`prop_value`* must be enclosed in quotation marks.

Note: The hierarchy editor does not analyze the contents of modules brought into the design by using `verilogfile` properties. Such modules are processed as "black boxes."

Note: Using a `verilogfile` property to override a view works only when your netlists are written to the run directory. In AMS Designer, by default, netlists are not written to the run directory and so using a `verilogfile` property generates an error such as the following during design preparation.

```
Error: verilogfile property
    '/usr1/cds11752/alpha6/novhdltestdir/SAR_A2D/AMS_lib/samplehold.vams'
    on amslib.samplehold:module cannot be processed.
    Enable netlist to run directory feature to process verilogfile property.
```

The Port Mapping File

The port mapping file is one way to specify how you want the software to connect Verilog vector buses to SPICE buses[1]. The software creates a `portmap_files` subdirectory and writes default port mapping files to this location. You can edit these files according to your connection requirements and use the `sourcefile` property together with the `sourcefile_opts -portmap_file` property in your `prop.cfg` file to specify any customized Verilog-to-SPICE port mapping files.

See the following topics for details:

- Creating a Custom Port Mapping File on page 384
- Specifying a Port Mapping on page 385
- Using Port Mapping Files: Rules to Remember on page 386
- Using Port Mapping Files: An Example on page 387

1. See "Using Port Mapping Files: Rules to Remember" on page 386 for other ways.

Creating a Custom Port Mapping File

To create a custom port mapping file, do the following:

1. Start with a `prop.cfg` file in which you do not specify a port mapping file, such as:

```
// prop.cfg
cell mysub1
{
... // No portmap_file option specified
}
cell mysub2
{
... // No portmap_file option specified
}
```

2. Run `ncelab`.

   The elaborator creates port mapping files in the `portmap_files` subdirectory. For example, if you run `ncelab` from `/net/usr1/cdn/port_conn`, the program creates the port map files in `/net/usr1/cdn/port_conn/portmap_files`:

```
/net/usr1/cdn/port_conn/portmap_files/mysub1.pb
/net/usr1/cdn/port_conn/portmap_files/mysub2.pb
```

3. Edit these port mapping files according to your connection requirements.

   See "Specifying a Port Mapping" on page 385 for information about how to format your port mapping statements.

4. Run ncelab again using `-portmap_file` to specify your custom port mapping files. For example:

```
// prop.cfg
cell mysub1
{
...
  string prop sourcefile_opts="-portmap_file /net/usr1/cdn/port_conn/
portmap_files/mysub1.pb";
}
cell mysub2
{
...
  string prop sourcefile_opts="-portmap_file /net/usr1/cdn/port_conn/
portmap_files/mysub2.pb";
}
```

Specifying a Port Mapping

The port mapping 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 mappings 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 mapping file:

- Port names match exactly (name-to-name), including casing
- Bus delimiters are `[]` and `<>`

Note: These default rules are the same as those that apply when you specify the `-auto_bus` property value with `sourcefile_opts`.

In the following example, `-case_map upper` changes all names to uppercase and `-bus_delim _` specifies the underscore character as a bus delimiter. The contents of the port mapping file (`analog_top.pb`) define the overriding binding rules (SPICE node `IN1` maps to Verilog input port `IN1`; SPICE node `in2` maps to Verilog input port `IN2`).

```
// prop.cfg
cell ANALOG_TOP
   {
      string prop sourcefile="analog_top.cir";
      string prop sourcefile_opts="-case_map upper -bus_delim _
                                   -portmap_file analog_top.pb";
   }
// analog_top.pb -- port mapping file
IN1 : IN1 dir=input
in2 : IN2 dir=input
```

Using Port Mapping Files: Rules to Remember

Here are some rules to remember when using port mapping files:

- The location where `ncelab` generates the port mapping files is

  `runDir``/portmap_files`

  where *`runDir`* is the directory where you run `ncelab`.
- If a port mapping file for a certain subcircuit already exists in the default location (mentioned in the previous bullet), `ncelab` will not overwrite the file and it will not generate a port mapping file for this case.
- The elaborator will never use a port mapping file unless you specify it explicitly in the `prop.cfg` file using `-portmap_file` option. A message from `ncelab` will clearly indicate whether the elaborator used a port mapping file.
- The file name in the argument of `-portmap_file` option must have a valid UNIX path, either absolute or relative to the directory where you run `ncelab`.

You can specify how to connect Verilog vector buses to SPICE buses using one or both of the following methods:

- Use the `sourcefile` property together with either the `sourcefile_opts -portmap_file` property to specify a Verilog-to-SPICE port mapping file (as discussed here) or the `sourcefile_opts -veri_file` property to specify a file containing a Verilog module that defines the port mappings to use from a Verilog parent to a SPICE subcircuit instance (as discussed in "The Verilog File for Port Mapping" on page 389)
- Use the `sourcefile` property together with the other `sourcefile_opts` properties

Rules you specify in a port mapping file or in a Verilog module that defines port mappings take precedence over any options you specify explicitly using other `sourcefile_opts` properties such as `-auto_bus`, `-case_map`, and `-bus_delim`.

Using Port Mapping Files: An Example

This example shows how you can use a port mapping file to define port mapping between Verilog and SPICE blocks in a very general way. Study the port mapping file format carefully to understand how you can create custom port mappings 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
```

Your `prop.cfg` file might contain the following:

```
cell analog_top
{
  string prop sourcefile="analog_top.cir";
  string prop sourcefile_opts="-portmap_file analog_top.pb"; }
```

Subcircuit `analog_top` (in `analog_top.cir`) might look like this:

```
.subckt analog_top
+ in1 itune[0] itune[1] in2

...
.ends analog_top
```

If your port mapping 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 mappings:

1. Port `in1` of SPICE subcircuit `analog_top` connects to the net 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 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`.

The Verilog File for Port Mapping

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 the `sourcefile` property together with the `sourcefile_opts -veri_file` property in your `prop.cfg` file. The `-veri_file` approach lets you replace the interface of a subcircuit with that of a Verilog module.

For example:

```
cell analog_top {
  string prop sourcefile="analog_top.cir";
  string prop sourcefile_opts="-veri_file analog_top.v";
}
```

The software applies port mappings (interfaces) defined in `analog_top.v` to instances of `analog_top`. While the elaborator uses the port mappings 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.cir`.

For example, `analog_top.v` might contain the following:

```
module  analog_top (in1, itune, in2);
inout in1;
inout [0:1] itune;
inout in2;

analog begin
end
endmodule
```

The `analog_top` subcircuit in `analog_top.cir` 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
```

the elaborator reads the interface you defined in module `analog_top` (in `analog_top.v`) and uses it to connect instance `analog_top` in module `top` as follows:

- Port `in1` and `in2` are scalar ports of direction `inout`.
- Port `itune` is a vector `inout` port.

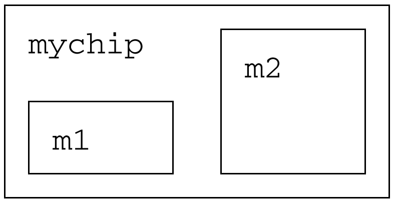
The elaborator uses this information to make the following connections:

- `pll_clk` in module `top` connects to SPICE port `in2`
- `itune[0]` in module `top` connects to SPICE port `itune[0]`
- `itune[1]` in module `top` connects to SPICE port `itune[1]`
- `ext_clk` in module `top` connects to SPICE port `in1`

Using hdl.var and cds.lib to Map Libraries and Views

The following example illustrates the concepts introduced in this chapter.

A file called `mychip.vams` contains a Verilog-AMS module called `mychip`. Module `mychip` instantiates two other modules: `m1` and `m2`.



You have two descriptions each of `m1` and `m2` as follows:

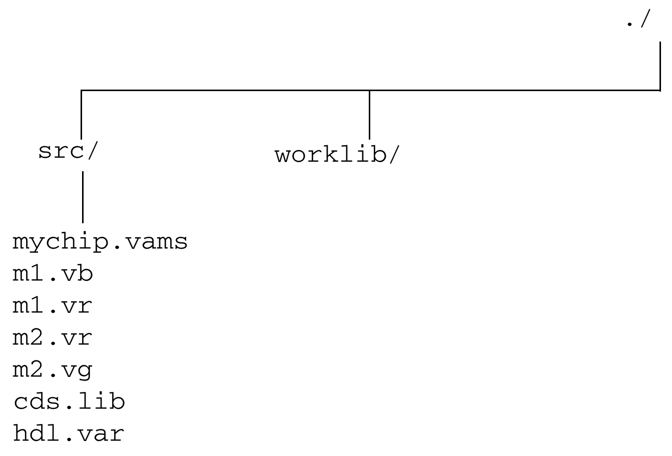
- For `m1`, you have both a behavioral and an RTL description in files `m1.vb` and `m1.vr`, respectively.
- For `m2`, you have both an RTL and a synthesized gate-level representation in files `m2.vr` and `m2.vg`, respectively.

Cell	Files	View
`mychip`	`mychip.vams`	Structural
`m1`	`m1.vb`	Behavioral
	`m1.vr`	RTL
`m2`	`m2.vr`	RTL
	`m2.vg`	Gates

You run your Cadence software from a directory called `src`. Your `src` directory contains the following files:

- All the design source files mentioned above
- A `cds.lib` file
- An `hdl.var` file

You create a `worklib` directory at the same level as `src` to use as the work library. Your directory structure looks like this:



Your `cds.lib` file contains the following statement, which defines a library called `worklib`:

```
# cds.lib file
DEFINE worklib ../worklib
```

Your `hdl.var` file contains definitions of the `LIB_MAP` and `VIEW_MAP` variables, which specify the library and view mapping for Verilog-AMS design units.

Note: The `LIB_MAP` and `VIEW_MAP` variables do not apply to VHDL.

```
# Define library mapping.
# Compile all files in src/ into worklib

DEFINE LIB_MAP (./ => worklib)

# Define view mapping.
# Files with .vb extension are compiled into view beh
# Files with .vr extension are compiled into view rtl
# Files with .vg extension are compiled into view gates
# Files with .vams extension are compiled into view module

DEFINE VIEW_MAP (.vb => beh, \
                 .vr => rtl, \
                 .vg => gates, \
                 .vams => module)
```

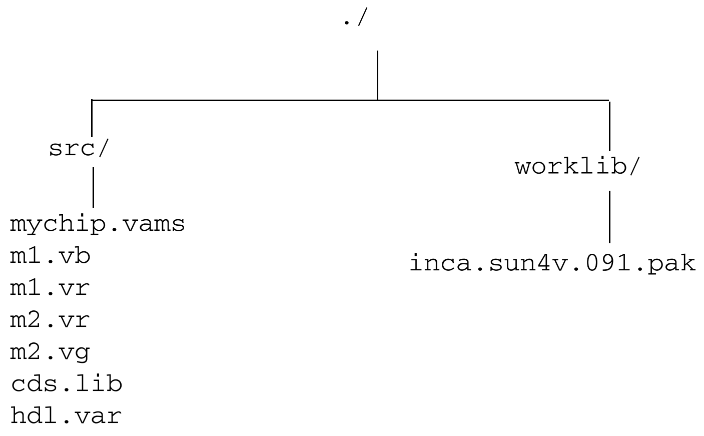
You use `ncvlog` to compile the design units in the source files as follows:

```
ncvlog mychip.vams m1.vb m1.vr m2.vg m2.vr
```

Using the mappings in `hdl.var`, the compiler creates a cell and view for each design unit. The library in this example is `worklib`. The design units in this example yield the following Library.Cell:View items:

Input File	Library.Cell:View
`mychip.vams`	`worklib.mychip:module`
`m1.vb`	`worklib.m1:beh`
`m1.vr`	`worklib.m1:rtl`
`m2.vg`	`worklib.m2:gates`
`m2.vr`	`worklib.m2:rtl`

The compiler creates a .pak file in the library directory (`worklib`) containing all of the intermediate objects.



You pass the Library.Cell:View notation of the top-level module (`mychip`) to `ncelab` to elaborate the design as follows:

```
ncelab worklib.mychip:module
```

Because there is only one view of module `mychip` in the library, you may omit the library and view specification as follows:

```
ncelab mychip
```

The elaborator generates a simulation snapshot for the design and stores intermediate objects in the `.pak` file. The snapshot is also a Library.Cell:View which, in this example, is called `worklib.mychip:module`.

You pass the elaboration snapshot to `ncsim` to simulate the design as follows:

```
ncsim worklib.mychip:module
```

Because there is only one snapshot in the library, you may omit the library and view specification as follows:

```
ncsim mychip
```

13 Compiling

After writing or editing your source files, the next step is to analyze and compile them. The program that you use to analyze and compile Verilog®-AMS source is called `ncvlog`. `irun` runs `ncvlog` automatically during the compilation phase.

`ncvlog` performs syntactic and static semantic checking on the HDL design units (modules, macromodules, or UDPs). If no errors are found, compilation produces an internal representation for each HDL design unit in the source files. These intermediate objects are stored in a library database file in the library directory.

You run `ncvlog` with options and one or more source file names. The arguments can be used in any order provided that option parameters immediately follow the option they modify. You can also run `ncvlog` with the `-unit` or `-specificunit` option and a design unit name.

To run `ncvlog` on Verilog-AMS modules, be sure to use the `-ams` option.

For example

```
ncvlog -ams foo.v foo2.vms          // foo.v and foo2.vms are source files
ncvlog -ams -unit worklib.mymod     // mymod is an HDL design unit
```

Note: If you are running `irun`, read the important note about using `-ams` in "irun Command Syntax" on page 280.

`ncvlog` treats each command-line argument that is not an option or a parameter to an option as a filename. For each filename, `ncvlog` first tries to open the file as specified. If this fails, each file extension specified with the `VERILOG_SUFFIX` variable is appended to the name, and `ncvlog` tries to open the file. The default file extension is `.v`. If no match is found, `ncvlog` tries the list of possible suffixes in the `hdl.var` variable `VIEW_MAP`. If all suffixes are exhausted, `ncvlog` issues an error.

For details on the `VERILOG_SUFFIX` and `VIEW_MAP` variables, see "hdl.var Variables" in the "Compiling Verilog Source Files" book.

`ncvlog` compiles each design unit into a Library.Cell:View. See "The Library.Cell:View Approach" on page 336 for information on the Virtuoso® AMS Designer simulator library system.

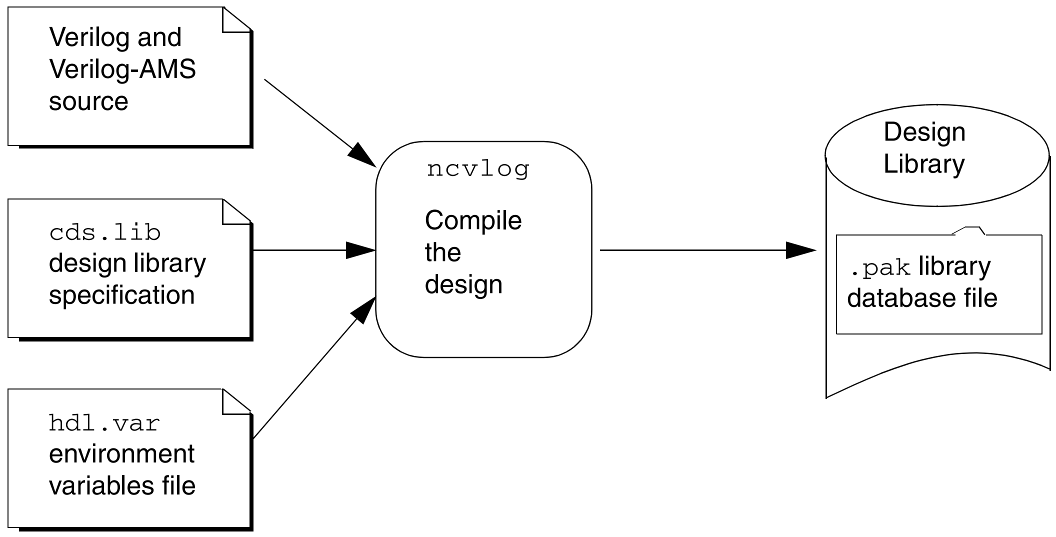
`ncvlog` always sets the cell name to the name of the design unit, but you can specify the library where the compiler stores compiled objects and the view names that are assigned to them. See "Controlling the Compilation of Design Units into Library.Cell:View" on page 403 for more information.

In addition to the intermediate objects for each HDL design unit, `ncvlog` can generate three other files: `master.tag`, `verilog.v`, and `pc.db`. These files are required if you want to

- Browse libraries using Cadence's Virtuoso® Design Environment software
- Use configurations

To generate these files, specify the `-use5x` option or define the `NCUSE5X` variable in the `hdl.var` file.

The following figure illustrates the `ncvlog` process flow:


Verilog and Verilog-AMS source
cds.lib design library specification
hdl.var environment variables file
ncvlog
Compile the design
Design Library
.pak library database file

ncvlog Command Syntax and Options

This section briefly describes the syntax and options for the `ncvlog` command. For more information, see the "Compiling Verilog Source Files" book.

```
ncvlog [options] filename { filename }
ncvlog [options] { -specificunit [Lib.]Cell[:View] filename |
                  -unit [Lib.]Cell[:View] }
```

ncvlog Command Option	Effect
`-AMs`	Enables analysis of Verilog-AMS design units. For more information, see "-AMs option" on page 399.
`-APpend_log`	Appends log data from multiple runs of `ncvlog` to one log file.
`-CDS_IMPLICIT_TMPDir` *`implicitTmpDir`*	Writes the compiled design data to *`implicitTmpDir`*, the implicit TMP directory for all libraries in the design. For more information, see "-CDS_IMPLICIT_TMPDir option" on page 400.
`-CDS_IMPLICIT_TMPOnly`	When you use this option with the `-update` option, the update operation looks only at design data in the *`implicitTmpDir`* you specify using the `-CDS_IMPLICIT_TMPDir` option. If you do not use this option, the update operation also considers design data it finds in libraries defined in `cds.lib` files. You can use the `-CDS_IMPLICIT_TMPOnly` option only when you also use the `-CDS_IMPLICIT_TMPDir` option.
`-CDSLib` *`cdslib_pathname`*	Specifies the `cds.lib` file to use.
`-CHecktasks`	Checks that all $tasks are predefined system tasks.
`-Define` *`identifier`*[**=***`value`*]	Defines a macro. For more information, see the "Defining Macros on the Command Line" section, in the "Compiling Verilog Source Files" book.
`-Errormax` *`integer`*	Specifies the maximum number of errors to process.
`-File` *`arguments_filename`*	Specifies a file of command-line arguments for `ncvlog` to use.

ncvlog Command Option	Effect
`-HDlvar` *`hdlvar_pathname`*	Specifies the `hdl.var` file to use.
`-HElp`	Displays a list of the `ncvlog` command-line options.
`-IEee1364`	Reports errors according to the IEEE 1364 Verilog standard.
`-INcdir` *`directory`*	Specifies an include directory.
`-LExpragma`	Enables processing of lexical pragmas.
`-LIBcell`	Marks all cells with `` `celldefine``.
`-LINedebug`	Enables line debug capabilities.
`-LOgfile` *`filename`*	Specifies the file to contain log data.
`-MEssages`	Turns on the printing of informative messages.
`-MOdelincdir` *`pathname`* {**`:`** *`pathname`*}	Specifies a list of paths to be searched for model files, included files, or files that are passed as instance parameter values.
`-NEverwarn`	Disables printing of all warning messages.
`-NOCopyright`	Suppresses printing of the copyright banner.
`-NOLIne`	Turns off source line locations for errors.
`-NOLOg`	Turns off generation of a log file.
`-NOMempack`	Enables use of the PLI routine `tf_nodeinfo()`, which is used to access memory array values.
`-NOPragmawarn`	Disables pragma-related warning messages.
`-NOStdout`	Suppresses the printing of most output to the screen.
`-NOWarn` *`warning_code`*	Disables printing of the specified warning message.
`-Pragma`	Enables pragma processing.
`-SPecificunit` [*`lib`*`.`]*`cell`*[**`:`***`view`*] *`filename`*]	Compiles the specified unit from the source file.
`-STatus`	Prints statistics on memory and CPU usage.
`-UNit` [*`lib`*`.`]*`cell`*[**`:`***`view`*]	Specifies the unit to be compiled.

ncvlog Command Option	Effect
-UPCase	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 `disciplines.vams` file that distinguishes `Voltage` nature from `voltage` discipline and `Current` nature from `current` discipline. In addition, using this option causes a clash between the `Force` nature and the `force` keyword.
-UPDate	Recompiles out-of-date design units. **Note:** The `-CDS_IMPLICIT_TMPOnly` option affects the behavior of the `-update` option.
-USe5x	Enables full 5x library system operation. You need full 5x library operation if you plan to use configurations.
-VErsion	Prints the compiler version number.
-VIew *view_name*	Specifies a view association.
-Work *library*	Specifies the library to be used as the work library.
-Zparse *SKILL_file*	Enables zparsing. The *SKILL_file* argument specifies the name of the SKILL file this option creates for importing Verilog-AMS text modules into the Virtuoso® design environment.

ncvlog Command Options Details

Most of the `ncvlog` command options are described in the “Compiling Verilog Source Files” book. This section describes only the `-ams` and `-cds_implicit_tmpdir` options.

-AMs option

Enables analysis of Verilog-AMS design units. Use this option to tell `ncvlog` that some or all of the HDL design units are written in the Verilog-AMS language. If you do not use this option, `ncvlog` 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 `ms10.v` and `ms12.v`, which both contain modules written with Verilog-AMS, you can use a command like

```
ncvlog -ams ms10.v ms12.v
```

Be careful to use the `-ams` 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.

-CDS_IMPLICIT_TMPDir option

Establishes a directory to hold any created temporary libraries. This directory is used even when `ASSIGN` statements in the `cds.lib` file specify different directories.

For example, your `cds.lib` contains

```
DEFINE amstestLib ./amstest
DEFINE basicLib ./basic
ASSIGN basicLib TMP ./basicTMP
DEFINE analogLib ./analog
ASSIGN AllLibs TmpRootDir ./myTMPs
```

Without a `-cds_implicit_tmpdir` option in effect, new data is written to

```
./myTMPs/amstest
./basicTMP/basic
./myTMPs/analog
```

Using the same `cds.lib` but with the option

```
-cds_implicit_tmpdir ./myImplTMPs
```

in effect, however, new data is written to

```
./myImplTMPs/amstest
./myImplTMPs/basic
./myImplTMPs/analog
```

Example ncvlog Command Lines

The following command includes the `-messages` option, which prints compiler messages.

```
ncvlog -messages 2bit_adder_test.v

ncvlog: v1.0.(p1): (c) Copyright 1995 - 2000 Cadence Design Systems, Inc.
file: 2bit_adder_test.v
        module worklib.top
                errors: 0, warnings: 0
%
```

The following example uses the `-work` option to define the current working library as `aludesign`. This overrides the definition of the `WORK` variable in the `hdl.var` file.

```
ncvlog -work aludesign 2bit_adder_test.v
```

The following example uses the `-file` option to include a file called `ncvlog.vc`, which includes a set of command-line options, such as `-messages`, `-nocopyright`, `-errormax`, and `-incdir`.

```
ncvlog -file ncvlog.vc 2bit_adder_test.v
```

In the following example, the `-ieee1364` option checks for compatibility with the IEEE specification. Error messages reference the *IEEE Language Reference Manual*.

```
ncvlog -ieee1364 2bit_adder_test.v
```

The following example includes the `-incdir` option to specify a directory to search for include files.

```
ncvlog -incdir ~larrybird/bigdesign 2bit_adder_test.v
```

In the following example, `-errormax 10` tells the compiler to stop compiling after 10 errors. The `-noline` option suppresses the reporting of source lines when errors are encountered. Using this option can improve performance when compiling very large source files that contain errors.

```
ncvlog -errormax 10 -noline 2bit_adder.v
```

The following example includes the `-logfile` option to send output to a log file called `adder.log` instead of to the default log file `ncvlog.log`.

```
ncvlog -messages -logfile adder.log 2bit_adder.v
```

The following example includes the `-linedebug` option, which disables the optimizations that prevent line debug capabilities.

```
ncvlog -linedebug 2bit_adder.v
```

The following example illustrates how to use the `-ams` options to compile Verilog-AMS module.

```
ncvlog -ams 2bit_adder.v
```

The following example illustrates using the -use5x option. You need to use this option if you plan to use a configuration when you elaborate.

```
ncvlog -ams -use5x top
```

hdl.var Variables

This section lists the `hdl.var` variables used by `ncvlog`. For more information, see “hdl.var Variables” in the “Compiling Verilog Source Files” book.

hdl.var Variables Used by ncvlog	Description
`LIB_MAP`	Maps files and directories to the names of libraries where you want them to be compiled.
`NCUSE5X`	Turns on generation of the `master.tag`, `verilog.v`, and `pc.db` files, which are required if you want to use the library browser using Cadence’s Virtuoso® Design Environment software. You also need these files if you plan to use configurations.
`NCVLOGOPTS`	Adds additional argument to the `ncvlog` command.
`SRC_ROOT`	Defines an ordered list of paths to search for source files when you are updating a design.
`VERILOG_SUFFIX`	Specifies valid file extensions for Verilog source files.
`VIEW`	Specifies the view name to use.
`VIEW_MAP`	Maps file extensions to view names.
`WORK`	Specifies the work library.

Conditionally Compiling Source Code

Use the conditional compilation compiler directives (`` `ifdef``, `` `else``, and `` `endif``) to include lines of an HDL source description conditionally during compilation. The `` `ifdef`` compiler directive checks whether a variable name is defined either in the source code or on the command line. If the variable name is defined, the compiler includes the lines in the source description.

For more information on these directives and on other directives that you can use for conditional compilation, see the “Controlling the Compiler” chapter of *Cadence Verilog-AMS Language Reference*.

Controlling the Compilation of Design Units into Library.Cell:View

When you run the `ncvlog` compiler, your HDL design units (modules, macromodules, and UDPs) are compiled into a Library.Cell:View. The cell name is always set to the name of the design unit, but you can control where the compiler stores compiled objects and the view names that are assigned to them.

To specify the library and view, you can use variables defined in the `hdl.var` file, command-line options, or compiler directives. The order of precedence is as follows (with cross-references to sections in the "Compiling Verilog Source Files" book).

1. By default, the library to compile into is the work library. See "The Default" section.
2. The definitions of the `LIB_MAP` and `VIEW_MAP` variables in the `hdl.var` file. See "The LIB_MAP and VIEW_MAP Variables".
3. The definitions of the `WORK` and `VIEW` variables in the `hdl.var` file. See "The WORK and VIEW Variables".
4. The `-work` or `-view` command-line options. See "The -work and -view Options".
5. The `-specificunit` command-line option with a library and/or view specification. See "The -specificunit Option".
6. The `` `worklib `` and `` `view `` compiler directives. See "The `worklib and `view Compiler Directives".

See "Mapping of Modules Defined Within `include Files" in the "Compiling Verilog Source Files" book for information on the library and view mapping of modules defined within `` `include `` files.

14

Elaborating

Before you can simulate your design, you must elaborate the design hierarchy. The `ncelab` 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

Note: `irun` runs `ncelab` automatically during the elaboration phase.

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 ncelab Process" on page 406.

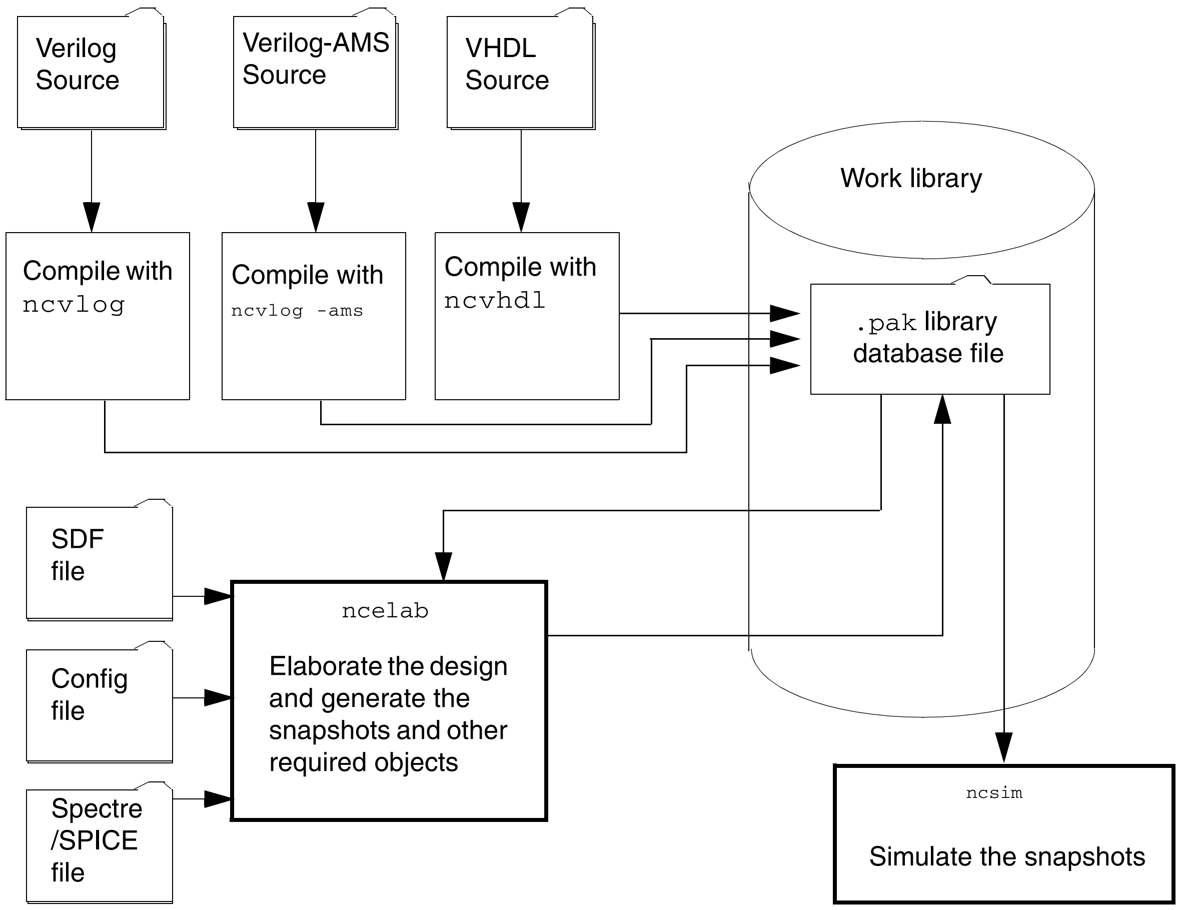
Note: The elaborator supports the occurrence-based binding feature of the Cadence hierarchy editor. For more information, see "Defining Rules at the Occurrence Level" in the *Cadence Hierarchy Editor User Guide*.

See the following related topics:

- Specifying the ncelab Command on page 407
- Using hdl.var Variables with ncelab on page 436
- Using the Simulation Front End (SFE) Parser on page 437
- Binding during Elaboration on page 442
- Enabling Read, Write, or Connectivity Access to Digital Simulation Objects on page 443
- Selecting a Delay Mode on page 444
- Setting Pulse Controls on page 445

Illustrating the ncelab Process

The following figure illustrates the `ncelab` process.



Specifying the ncelab Command

The syntax for the ncelab command is as follows:

ncelab [*options*] [*Lib*.]*Cell*[:*View*] { [*Lib*.]*Cell*[:*View*] }

See also

- Specifying Design Units for Elaboration on page 418
- ncelab Command Option Details on page 419

ncelab Command Option	Description
-ACCEss [+] [-] *access_spec*	Sets the visibility access for all objects in the design.
-AFile *access_file*	Specifies an access file.
-AMSFastspice	Uses the UltraSim solver. When you use this option ■ The ports of SPICE and Verilog-A instances in the sections of the design that are simulated by the UltraSim solver are always taken as `electrical`. The direction of such ports is always taken as `inout`. ■ The disciplines of nets used in the sections of the design that are simulated by the UltraSim solver are not checked for discipline incompatibility.
-AMSInput *spectre_file*	Specifies a Spectre-language file. You can specify more than one Spectre-language file on the `ncelab` command line: `ncelab -amsi file1.scs -amsi file1.scs`
-AMSPARTINFO *part_file*	Specifies a file to hold mixed-signal partition and connect module insertion information. The file also contains information about Real Number Modeling net type (for example, `wrealavg`, `wrealsum` and so on) in the design. The format of the file might change from release to release.
-amssie	See Using the Strength-Based Interface Element (SIE) on page 228 for more information.

ncelab Command Option	Description
-amselabtrace ssoveride	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.
-ANno_simtime	Enables the use of PLI/VPI routines that modify delays at simulation time.
-APpend_log	Appends log information from more than one run of `ncelab` to one log file.
-Binding [*lib*.]*cell*[:*view*]	Forces an explicit submodule binding.
-CDS_IMPLICIT_TMPDir *implicitTmpDir*	Specifies an implicit TMP directory to search for design data and to hold new design data. The software writes this option to the snapshot header for later use by the simulator and `ncupdate`.
-CDS_IMPLICIT_TMPOnly	When you use this option with the `-update` option, the update operation looks only at design data in the *implicitTmpDir* you specify using the `-CDS_IMPLICIT_TMPDir` option. If you do not use this option, the update operation also considers design data it finds in libraries defined in `cds.lib` files. You can use the `-CDS_IMPLICIT_TMPOnly` option only when you also use the `-CDS_IMPLICIT_TMPDir` option. The software writes this option to the snapshot header for later use by the simulator and `ncupdate`.
-CDslib *cdslib_pathname*	Specifies the `cds.lib` file to use.
-CHkdigdisp	Checks the compatibility of objects with different discrete disciplines. For more information, see "-chkdigdisp Option" on page 419.
-COverage	Enable coverage instrumentation.
-DELay_mode {**zero** | **unit** | **path** | **distributed** | **none**}	Specifies the delay mode to use for digital Verilog-AMS portions of the hierarchy.

ncelab Command Option	Description
`-DESktop`	Specifies that you want to use the desktop simulator after starting the launch program.
`-DISCipline` *`discipline_name`*	Specifies the discipline of discrete nets for which a discipline is otherwise undefined. For more information, see "-discipline Option" on page 420.
`-DISRes default|detailed|none`	Specifies the kind of discipline resolution to use or turns off discipline resolution completely. For more information, see "-disres Option" on page 420.
`-EPULSE_NEg`	Filters cancelled events (negative pulses) to the e state.
`-EPULSE_NOneg`	Does not filter cancelled events (negative pulses) to the e state.
`-EPULSE_ONDetect`	Uses On-Detect filtering of error pulses.
`-EPULSE_ONEvent`	Uses On-Event filtering of error pulses.
`-ERrormax` *`integer`*	Specifies the maximum number of errors to process.
`-EXpand`	Expands all vector nets.
`-File` *`arguments_filename`*	Specifies a file of command-line arguments for `ncelab` to use.
`-GATELOOPWARN`	Detects zero-delay loop of gate-level Verilog (or VHDL) models and prints the details in the form of proper error messages.
`-GENAfile` *`access_filename`*	Generates an access file with the specified file name.
`-GENEric` *`generic_name`* **`=>`** *`value`*	Specifies a value for a top-level generic.
`-HDlvar` *`hdlvar_pathname`*	Specifies the `hdl.var` file to use.
`-HElp`	Displays a list of `ncelab` command-line options with a brief description of each option.
`-IEee1364`	Checks for compatibility with the IEEE 1364 standard.

ncelab Command Option	Description
-IEReport/-IEInfo	Generates a detailed report on interface elements. For more information, see "-iereport/-ieinfo Option" on page 421.
-INtermod_path	Enables multisource and transport delay behavior with pulse control for interconnect delays.
-LIBVerbose	Displays messages about module and UDP instantiations.
-LOADPli1 *shared_library_name*:*bootstrap_function_name*	Dynamically loads the specified PLI 1.0 application.
-LOADVpi *shared_library_name*:*bootstrap_function_name*	Dynamically loads the specified VPI application.
-LOGfile *filename*	Specifies the file to contain log data.
-MAxdelays	Applies the maximum delay value from a timing triplet in the form min:typ:max in the SDF file while annotating to Verilog or to VITAL.
-MEssages	Prints informative messages during simulation.
-MIndelays	Applies the minimum delay value from a timing triplet in the form min:typ:max in the SDF file while annotating to Verilog or to VITAL.
-mixed_bus_opt	Does not allow mixed buses to be automatically generated for unsupported constructs.
-MIXesc	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.
-MODELIncdir *pathname* {: *pathname*}	Specifies a list of directories to be searched for model files, included files, or files that are passed as instance parameter values. For more information, see "-modelincdir Option" on page 422.

ncelab Command Option	Description
-MODELPath "[**cell-**[*lib_name*.]*cell_name*[**:***view_name*]**-**] *pathname* [**(***section***)**]{**:** *pathname* [**(***section***)**]}**"**	
	Specifies SPICE or Spectre source files for the models to be used in a specified scope and in scopes below the specified scope. If a source file is a library file (which begins with the `library` keyword), you must specify a *section*. For more information, see "-modelpath Option" on page 423.
-NEG_tchk	Allows negative values in `$setuphold` and `$recrem` timing checks in the Verilog description and in `SETUPHOLD` and `RECREM` timing checks in SDF annotation.
-nettype_port_relax	Allows for relaxed port compatibility rules for connections of built-in net types. See Using Real Number Modeling in SystemVerilog on page 257 for more information.
-NEVerwarn	Disables printing of all warning messages.
-NO_Sdfa_header	Turns off the printing of elaborator information messages that display information contained in the SDF command file.
-NO_TCHK_Msg	Turns off the display of timing check warning messages.
-NO_TCHK_Xgen	Turns off X generation in accelerated VITAL timing checks.
-NO_VPD_Msg	Turns off glitch messages from accelerated VITAL pathdelay procedures.
-NO_VPD_Xgen	Turns off X generation in accelerated VITAL pathdelay procedures.
-NOAutosdf	Turns off automatic SDF annotation.
-NOCopyright	Suppresses printing of the copyright banner.
-NOIpd	Turns off recognition of input path delays in a VITAL level 1 cell and uses the non-delayed input signals directly.

ncelab Command Option	Description
`-NOLog`	Turns off generation of a log file.
`-NONotifier`	Tells the elaborator to ignore notifiers in timing checks.
`-NOPAramerr`	Tells the elaborator to allow undeclared parameters to be overridden. For more information, see "-noparamerr Option" on page 424.
`-NOPOrterr` (scope) *`port_name`*	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 "-noporterr Option" on page 424.
`-NOSOurce`	Turns off source file timestamp checking when using the `-UPdate` option.
`-NOSTdout`	Suppresses the printing of most output to the screen.
`-NOTimingchecks`	Turns off the execution of timing checks.
`-NOVitalaccl`	Suppresses the acceleration of VITAL level 1-compliant cells.
`-NOWarn` *`warning_code`*	Disables printing of the specified warning message.
`-NTc_warn`	Print convergence warnings for negative timing checks for both Verilog and VITAL if delays cannot be calculated given the current limit values.
`-OMicheckinglevel` *`checking_level`*	Specifies the OMI checking level to use.
`-PAthpulse`	Enable `PATHPULSE$` specparams, which are used to set module path pulse control on a specific module or on specific paths within modules.
`-PLINOOptwarn`	Prints a warning message only the first time that a PLI read, write, or connectivity access violation is detected.
`-PLINOWarn`	Disables printing of PLI warning and error messages.
`-PReserve`	Preserves resolution functions on signals with only one driver.

ncelab Command Option	Description
-PROpspath *path*	Specifies the path to the `prop.cfg` file when you do not use `config` settings in the Cadence hierarchy editor (HED). If you use `-propspath` to specify a property file, you must not also use the property columns available in the HED.
-PULSE_E *error_percent*	Sets the percentage of delay for the pulse error limit for both module paths and interconnect.
-PULSE_INT_E *error_percent*	Sets the percentage of delay for the pulse error limit for interconnect only.
-PULSE_INT_R *reject_percent*	Sets the percentage of delay for the pulse reject limit for interconnect only.
-PULSE_R *reject_percent*	Sets the percentage of delay for the pulse reject limit for both module paths and interconnect.
-Relax	Enable relaxed VHDL interpretation.
-SCope_discipline *disc_scope*	Specifies a suggested discipline for the nets in a specified scope. For more information, see "-scope_discipline Option" on page 426
-SDF_Cmd_file *sdf_command_file*	Specifies an SDF command file to control SDF annotation.
-SDF_NO_Errors	Suppresses error messages from the SDF annotator.
-SDF_NO_Warnings	Suppresses warning messages from the SDF annotator.
-SDF_NOCheck_celltype	Disables celltype validation between the SDF annotator and the Verilog description.
-SDF_Precision *argument*	SDF data is modified to this precision.
-SDF_Verbose	Includes detailed information in the SDF log file.
-SDF_Worstcase_rounding	For timing values in the SDF file, truncates the `min` value, rounds the `typ` value, and rounds up the `max` value.

ncelab Command Option	Description
-SEtdiscipline [**no_dr**] *disc_scope disc_name*	Specifies a discipline for the elaborator to use for domainless nets in the specified scope. Using `no_dr` turns off discipline resolution in the specified scope and for the entire block. See "-setdiscipline Option" on page 427.
-SNapshot *snapshot_name*	Specifies a name for the simulation snapshot. If you do not use this option, `ncelab` 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.
-SPECTRE_Argfile_spp *filename*	Specifies the path to a file containing space-separated command arguments for `spp`. You do not need this option if you are using the simulation front end (SFE) parser.
-SPECTRE_E	Runs the Spectre parser with the `-E` option (so that the C preprocessor runs) when parsing files specified by the `-MODELPath` option.
-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.
-STatus	Prints statistics on memory and CPU usage after elaboration.
-TImescale '*time_unit* **/** *time_precision*'	Sets the default timescale for Verilog (digital) modules that do not have a timescale set.
-TYpdelays	Applies the typical delay value from a timing triplet in the form min:typ:max in the SDF file while annotating to Verilog or to VITAL.
-UPdate	Recompiles out-of-date design units and then re-elaborates the design. **Note:** The `-CDS_IMPLICIT_TMPOnly` option affects the behavior of the `-update` option.

ncelab Command Option	Description
`-USe5x4vhdl`	Specifies that configurations apply to VHDL as well as Verilog-AMS and that configurations take precedence over VHDL default binding and other searches. For more information, see "-use5x4vhdl Option" on page 433.
`-V93`	Enables VHDL-93 features.
`-VErsion`	Prints the version of the elaborator and exits.
`-VIPDMAx`	Selects the Max. delay value for VitalInterconnectDelays.
`-VIPDMIn`	Selects the Min. delay value for VitalInterconnectDelays.
`-Work` *work_library*	Specifies the library to be used as a work library.
-`rnm_coerce`	

ncelab Command Option	Description
	Enables scope-based turning off of wreal coercion. 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. Possible values are: `none` - disable wreal coercion. `detailed` - enable wreal coercion. `default` - enable global coercion with default resolution. `off scopeType` *`scope-`* disable local coercion in scope, coercion in other scope is ON. If you are a digital-centric user running an AMS simulation that requires only the digital solver, we recommend specifying `-rnm_coerce none`. Example: `rnm_coerce "off inst-top.dcinst-"` All the net of instance `top.dcinst` and its children will not be coerced to wreal; top level and other instances will be coerced as normal.

ncelab Command Option	Description
`-wreal_resolution` *`resFunc`*	Specifies the `wreal` resolution function you want the elaborator to use. Valid Values: ■ `default` – Default setting ■ `fourstate` – Verilog 4-state logic resolution algorithm ■ `sum` – Summation of all drivers ■ `avg` – Average of all drivers ■ `min` – Minimum value of all drivers ■ `max` – Maximum value of all drivers See “Selecting a wreal Resolution Function” on page 244 for more information.

The *`Lib`*, *`Cell`*, and *`View`* arguments identify the top-level cells. You can specify the options and the arguments in any order provided that the parameters to an option immediately follow the option.

- You must specify at least the cell for each of the top-level units.
- If a top-level cell with the same name exists in more than one library, Cadence recommends you specify the library also on the command line.
- If there are multiple views of the top-level units, Cadence recommends you specify the view on the command line. If you do not specify the view, `ncelab` uses the following rules to resolve the reference to the top-level design unit:

  a. Search the library defined with the `WORK` variable in the `hdl.var` file. If one view of the cell exists in that library, use that view. Generate an error message if more than one view exists.

  b. If the `WORK` variable is not defined in the `hdl.var` file, search the libraries defined in the `cds.lib` file. If one view of the cell exists in the libraries, use that view. Generate an error message if more than one view exists.

Elaboration produces a simulation snapshot, which also has a Library.Cell:View name. Unless the `-SNapshot` option is used to explicitly name the snapshot, the parts of the Library.Cell:View name are as follows:

- *Library* is the name of the library where the top-level unit on the `ncelab` command line is found. If more than one Verilog top-level module is specified on the command line, the

Library is the name of the library where the first top-level module listed on the command line is found.

- *Cell* is the name of the top-level unit on the `ncelab` command line. If more than one Verilog top-level module is specified on the command line, the *Cell* is the name of the first top-level module listed on the command line.
- *View* is the view name that is specified for the first top-level design unit on the `ncelab` command line or (if a view is not specified) the name of the view that is used as a result of the rules that `ncelab` uses to resolve references to top-level units given on the `ncelab` command line.

See also

- ncelab Command Option Details on page 419
- Example ncelab Command Lines on page 435

Specifying Design Units for Elaboration

When you run `ncelab`, you specify command-line options and the Library.Cell:View name or names of the top-level HDL design units. You can specify any of the following design units on the command line:

- Exactly one VHDL top-level unit
- One or more Verilog® or Verilog-AMS top-level units
- Exactly one VHDL unit and one or more Verilog or Verilog-AMS units
- Exactly one configuration

In addition, especially if you are using the Virtuoso® AMS Designer environment, you might specify the following modules:

- `cds_globals`
- `connectrules`

You must not instantiate design units specified on the command line beneath themselves in the design because this practice results in recursive instantiations.

ncelab Command Option Details

For detailed information about `ncelab` command options, see the “Elaboration Command-Line Options” book. Information about the following command options is available here:

- -amsfastspice Option on page 419
- -chkdigdisp Option on page 419
- -discipline Option on page 420
- -disres Option on page 420
- -iereport/-ieinfo Option on page 421
- -modelincdir Option on page 422
- -modelpath Option on page 423
- -noparamerr Option on page 424
- -noporterr Option on page 424
- -propspath Option on page 426
- -scope_discipline Option on page 426
- -setdiscipline Option on page 427
- -use5x4vhdl Option on page 433
- -wreal_resolution Option on page 433

Note: You can also use these command options with `irun`.

-amsfastspice Option

Specifies that the elaborator is to prepare for using the UltraSim solver.

-chkdigdisp Option

After discipline resolution finishes, this option checks the compatibility of discrete disciplines, using the assumption that discrete disciplines with different names are incompatible. The check:

- Determines the disciplines of digital nets that do not already have disciplines by looking at the discrete disciplines of connected nets.

- Examines every digital island of mixed-signal nets. (A digital island includes all the digital nets that are directly connected to each other without passing through an analog net.)
- Assumes that VHDL nets encountered in the digital islands have `logic` discipline.

  Note: The `-chkdigdisp` option ignores the disciplines set for VHDL digital nets by using the `-setdiscipline` option, and always treats the VHDL digital nets as `logic` nets.

The elaborator does not check digital islands that are connected to a `supply0` net.

If the discrete discipline check fails, the elaborator issues an error message with information about the incompatible disciplines and the affected nets.

-discipline Option

Specifies a discipline for discrete nets for which a discipline is either not specified or cannot be determined through discipline resolution. For example, the following command line specifies that the `logic` discipline is to be used for nets that do not have a known discipline.

```
ncelab -discipline logic top
```

-disres Option

Specifies the discipline resolution method to use or turns off discipline resolution completely. For more information about these methods, see "Discipline Resolution Methods" in the *Cadence Verilog-AMS Language Reference*.

`-disres detailed`	causes the elaborator to use the detailed form of discipline resolution
`-disres default`	causes the elaborator to use the default form of discipline resolution
`-disres none`	turns off discipline resolution completely

For example, you can turn off discipline resolution in the entire design and specify the default discrete discipline `logic1` for domainless nets as follows:

```
irun -disres none -discipline logic1 ...
ncelab -disres none -discipline logic1 ...
```

if you have a SPICE or Verilog-A block that you introduce using a `prop.cfg` file or a modelpath setting, instantiate in a digital block, and in turn instantiate in the top-level design. You can use the `prop.cfg` file or modelpath setting to indicate that the SPICE or Verilog-A block is an `analog` block of the `electrical` discipline. The `-discipline logic1` option applies the `logic1` discipline to domainless nets in the digital block such that the boundary between the continuous and discrete domains remains clearly defined.

-iereport/-ieinfo Option

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 Name, CE report summary, and so on. For example:

```
1. 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 (electrical)
 Port :
testbench.msbuf@ms_buf<module>.I2@ana_nand<module>.I1@my_inv<module>.in (logic18V
input)
 Parameters :
         vsup : 1.8
         vthi : 1.2
         vtlo : 0.6
         tr : 0.0
 List of Ports connected to net testbench.msbuf.I2.Y1 : (Total: 1)
         testbench.msbuf.I2.I1.in (logic18V input)


2. Interface Elements at the block <instance> testbench of <master> testbench (file
: /home/bcui/BDR_multpwr/source/digital/testbench.v)
 Automatically inserted : testbench.msbuf_out__L2E__logic18V
 Connect Module : L2E
 Mode :          Merged
 Net :           testbench.msbuf_out (electrical)
 Port :          testbench.msbuf@ms_buf<module>.out (logic18V output)
 Parameters :
         vsup : 1.8
         vthi : 1.2
         vtlo : 0.6
         tr : 0.0
         tf : 0.0
         tx : 0.0
         tz : 0.0
         rlo : 200
         rhi : 200
         rx : 40
         rz : 10000000
 List of Ports connected to net testbench.msbuf_out : (Total: 1)
         testbench.msbuf.out (logic18V output)
---------- Conversion Element (CE) Report ----------
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 #2: Name:        MY_AD_LIB.ILOG2E:behavior
      File:        ILOG2E.vhms
      Instance:    :vh_top:test2:ilog2e_a
      Generic Map: ()
      VHDL Signal: input ':vh_top:A' with type 'ilog'
      Spice Node name: dummy_spice2.A
----  CE Report Summary:
E2ILOG ( ELECTRICAL inout; ILOG out; )         total: 1
ILOG2E ( ILOG in; ELECTRICAL inout; )  total: 1
------------------------------------------------------------------------
Total Number of Conversion Elements                          total: 2
Total Number of Optimized Conversion Elements                total: 0
Total Number of effective Conversion Element Instances       total: 2
```

Note: The `-iereport` option is aliased to the `-ieinfo` option.

-ieinfo_log

By default, the `-ieinfo` option writes the results in a file `ams_ieinfo.log`. You can use the `-ieinfo_log` option to specify a file to which the output of the `-ieinfo` option is written.

-modelincdir Option

Specifies a list of directories to be searched for model files, included files, or files that are passed as instance parameter values.

You can also achieve the same result by defining the `MODELINCDIR` variable in the `hdl.var` file. If you use both the `MODELINCDIR` variable and the `-modelincdir` option, the latter takes precedence. For more information, see “The hdl.var File” on page 347.

For example, you have a file with the following contents.

```
// model.scs
simulator lang=spectre
include "moremodels.scs"
ahdl_include "mymodel.va"
...
v1 (s gnd) vsource type=pwl file="wave.pwl"
...
```

In addition, `moremodels.scs` is in the `basemodels` directory and `mymodel.va` and `wave.pwl` are in the `detailmodels` directory. To ensure that all the models are found, you use a `-modelincdir` option, as follows:

```
ncelab -modelincdir $basedir/basemodels:$basedir/detailmodels top
```

-modelpath Option

Specifies SPICE or Spectre source files for the models to be used in a specified scope and in scopes below the specified scope. If a source file is a library file (which begins with the `library` keyword), you must specify a section also for that file. If the scope specification is omitted, the elaborator uses the global scope by default.

You can also achieve the same result by defining the `MODELPATH` variable in the `hdl.var` file. If you use both the `MODELPATH` variable and the `-modelpath` option, the latter takes precedence.

Note: You can also specify model files for an analog device using the `include` statement in the AMS control file.

For example, the file `simple_cap.m` contains the following definition. This file instantiates an analog model, `my_mod_cap`.

```
// cap model definition

simulator lang=spectre
parameters base=8

model  my_mod_cap capacitor c=2u tc1=1.2e-8 tnom=(17 + base)  w=4u l=4u cjsw=2.4e-10
```

You can use these definitions in a command like this one:

```
ncelab -modelpath "simple_cap.m" top
```

-noparamerr Option

By default, the elaborator reports an error and stops when it encounters a value override for an undeclared parameter. Specifying `-noparamerr` tells the elaborator to allow undeclared parameters to be overridden.

For example, the following command line permits overriding the values of undeclared parameters, such as by using a `defparam` statement or by overriding the value when an instance is declared.

```
ncelab -noparamerr top
```

-noporterr Option

By default, the elaborator reports an error and stops when it processes an instance that does not include all the ports that are specified in the port connection list. Specifying `-noporterr` tells the elaborator to allow such instantiations and to issue warnings instead of errors.

Specify the scope and *port_name* in one of the following ways:

"inst-*hier_instance_name***-** *port_name***"**

Denotes a particular hierarchical instance and all instances in the subhierarchies rooted at that instance.

"cell-*lib_name***.***cell_name***:***view_name***-** *port_name***"**

Denotes all instances of the given cellview of the given cell and library, and all additional instances of the given cellview found in the subhierarchies rooted at instances of the given cell.

"cell-*lib_name***.***cell_name***-** *port_name***"**

Denotes all cells of the given name from the given library and all instances in the subhierarchies rooted at those cells.

"cell-*cell_name***-** *port_name***"**

Denotes all cells of the given name and all instances in the subhierarchies rooted at those cells.

"lib-*library_name***-** *port_name***"**

Denotes all cells from the given library.

"cellterm-*cell_name***.***port_name***-** (*port_name*)**"**

Denotes all ports of the given name in all cells of the given name, but *not* instances in the subhierarchies rooted at those cells. The *port_name* adjacent to the period is required but the following *port_name*, being redundant, is optional and is ignored if given.

"instterm-*hierarchical_instance_name***.***port_name***-** (*port_name*)**"**

Denotes all ports of the given name in the given hierarchical instance, but not instances in the subhierarchies rooted at that instance. The *port_name* adjacent to the period is required but the following *port_name*, being redundant, is optional and is ignored if given.

With `-noporterr` in effect, the elaborator ignores connections-by-name to nonexistent ports (but issues a warning). The `-noporterr` does not affect connections-by-order to nonexistent ports and elaboration stops.

For example, consider the following modules which, without using `-noporterr`, result in many elaboration errors because of missing ports in the `child` and `grandchild` modules.

```
module top;
    wire a,b,vdd,vss;
    child c1(.x(a), .y(b), .vdd(vdd), .vss(vss));
    child c2(.x(a), .y(b), .vdd(vdd), .vss(vss));
endmodule
module child(x, y);
    input x, y;
    wire vdd, vss;
    grandchild g1(.a(a), .b(b), .vdd(vdd), .vss(vss));
    grandchild g2(.a(a), .b(b), .vdd(vdd), .vss(vss));
endmodule
module grandchild(a, b);
    input a, b;
endmodule
```

To work around the missing ports you can use the `-noporterr` option in the following ways.

- Providing the following options to `ncelab`,

  ```
  -noporterr vdd -noporterr vss
  ```

  causes the elaborator to ignore the incorrect `vdd` and `vss` port connections and issue warnings for all the missing connections.

- Providing the following option to `ncelab`,

  ```
  -noporterr "cell-child- vdd"
  ```

causes the elaborator to warn about and ignore the connection to the nonexistent port `vdd` in both instantiations of the `child` module, and ignore the connection to the nonexistent port `vdd` in all four instantiations of the `grandchild` module.

- Providing the following option to `ncelab`,

  ```
  -noporterr "instterm-top.c1.g1.vdd"
  ```

  causes the elaborator to warn about and ignore the connection to the nonexistent port `vdd` in the instantiation of `top.c1.g1`, but to continue to error out on all other missing connections.

- Providing the following option to `ncelab`,

  ```
  -noporterr "inst-top.c1 vss"
  ```

  causes the elaborator to warn about and ignore the connection to the nonexistent port `vss` in the instantiations of `top.c1`, `top.c1.g1`, and `top.c1.g2`

-propspath Option

Specifies the `prop.cfg` file to use when running the elaborator. For more information, see "The Property (prop.cfg) File" on page 365.

-scope_discipline Option

Specifies a suggested discipline for the digital nets (but not the analog nets) in a specified scope. The option does not apply to any hierarchy levels above the specified scope. The argument for the `-scope_discipline` option takes the following format.

"*type_of_object-hierarchical_name_of_object- digital_discipline*"

where *`type_of_object`* is `lib`, `cell`, `cellterm`, `inst`, `instterm`, or `net`.

If *`hierarchical_name_of_object`* is a vector, the *`digital_discipline`* is applied to all element of the vector.

The `-scope_discipline` option is not supported for mixed-language designs.

Cadence recommends using the `-setdiscipline` option instead because support for the `-scope_discipline` option might be withdrawn in the future.

For example,

```
ncelab -discipline logic5v -scope_discipline "inst-top.I1- logic"
```

sets the default discrete discipline for nets in instance `top.I1` to logic, but the default discrete discipline for nets in the rest of the design is `logic5v`.

The next example,

```
ncelab -discipline logic -scope_discipline "inst-top.I1- logic5v"
                         -scope_discipline "inst-top.I1.I2- logic"
```

sets the suggested discipline for nets in instance `top.I1.I2` to `logic`, which is the same value as the default value for nets in the design as a whole.

-setdiscipline Option

Specifies a continuous or discrete discipline you want the elaborator to use for domainless nets in a specified scope (and in scopes below the specified scope). Using `no_dr` turns off discipline resolution in the specified scope and for the entire block but allows discipline resolution to run in other scopes of the design.

You can use `-setdiscipline` in mixed-language designs (Verilog-AMS and VHDL-AMS). When you use `-setdiscipline` in a mixed-language design, the elaborator applies your settings to the specified Verilog-AMS and VHDL digital nets. (Discipline settings do not apply to VHDL analog nets that already have natures specified using `MAPN2D` in your `hdl.var` file.) For Verilog-AMS scope, matching rules are case-sensitive. For VHDL-AMS scope, matching rules are case-insensitive.

Specify the *`disc_scope`* and *`disc_name`* in one of the following ways:

```
"NET-hierarchical_net_name- discipline_name"
"INSTTERM-hierarchical_port_name- discipline_name"
"INST-hierarchical_instance_name- discipline_name"
"CELLTERM-lib_name.cell_name.port_name- discipline_name"
"CELLTERM-cell_name.port_name- discipline_name"
"CELL-lib_name.cell_name:view_name- discipline_name"
"CELL-lib_name.cell_name- discipline_name"
"CELL-cell_name- discipline_name"
"LIB-lib_name- discipline_name"
```

Note: The most detailed rule has the highest precedence. The forms listed above appear in precedence order from highest to lowest. The top two forms (`INSTTERM` and `NET`) have the same (highest) precedence value.

See also “Notes on Specifying Disciplines for Mixed Verilog-AMS/VHDL-AMS Designs” on page 432.

Here are some examples:

```
-setdiscipline "INST-top.f1.f2.f3.f4- logic"
-setdiscipline "cell-foo- logic"
-setdiscipline "cellterm-foo.p- electrical"
```

Tip

You can use a wildcard character at the end of the scope name to specify more than one scope at a time.

If the elaborator does not find the specified discipline in the design, it creates a discrete discipline with the specified discipline name and applies that new discipline to domainless nets.

In the following example, the top-level design unit, `Top`, contains digital block `D1`, which in turn contains a Verilog-AMS analog block `A1`. The `-setdiscipline` statements turn off discipline resolution in blocks `D1` and `A1`, and specify `logic1` as the discrete discipline for domainless nets in `D1` and `electrical` as the continuous discipline for domainless nets in `A1`. These settings restrict the upward propagation of the continuous domain from the analog block (typically a SPICE block) that would otherwise occur when there are domainless nets in `D1` that connect to `A1`.

```
-setdiscipline "no_dr inst-top.D1- logic1"
-setdiscipline "no_dr inst-top.D1.A1- electrical"
```

When you use `-setdiscipline "no_dr"` in a mixed Verilog-AMS/VHDL-AMS design, the elaborator performs a discipline domain compatibility check inside VHDL-AMS blocks as well, such that you cannot set an analog discipline on a net that connects to a VHDL-AMS block that contains a digital signal.

Precedence of Discipline Specification Methods

When disciplines are applied to nets in more than one way, the elaborator determines the effective value by using the following rules of precedence.

Discipline Specification Method	Precedence

Explicitly defining the discipline. For example,
`module example2;`
`    electrical net;`
`endmodule`
	Highest precedence

Overriding a discipline with out-of-module references (for domainless nets). For example,
`module example1;`
`    electrical example2.net;`
`endmodule`
	↓

Defining disciplines with the `-setdiscipline` option (for domainless nets). For example, `-setdiscipline "no_dr inst-top.d1- logic1"`	↓
Obtaining disciplines through the discipline resolution process (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
`ncelab -discipline logic`
or
`ncelab -scope_discipline "inst-top.d1- logic2"`
used on the command line, both specify a global default logic discipline for domainless nets that you can override using methods shown higher in the chart.
	Lowest precedence

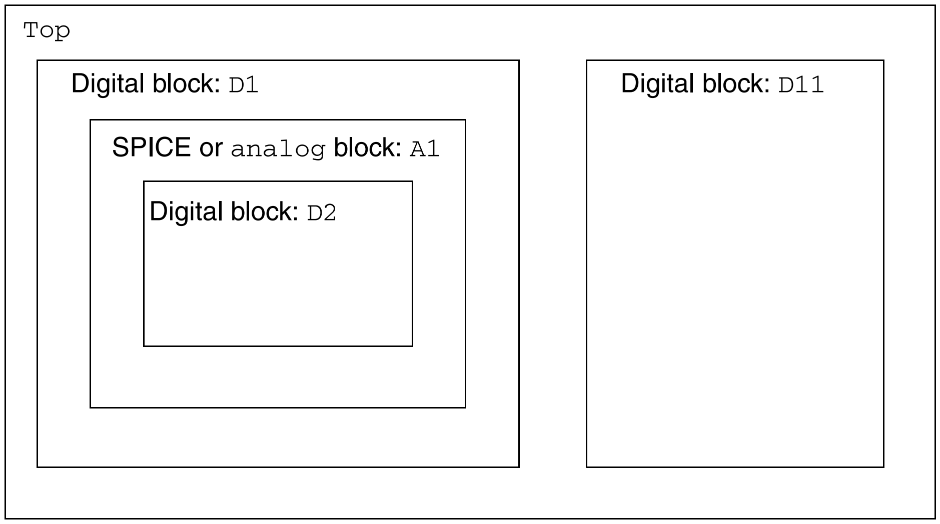
Note: The `` `default_discipline`` compiler directive is applied at analysis (`ncvlog`) stage to the source on which it applies. During elaboration, since the source is already analyzed, there is no way to inherit the source-level settings from the instantiating scope. The same behavior exists for other compiler directives like `` `default_nettype ``. If you want to control the discipline resolution hierarchically, use the `-setdiscipline` option.

Guidance for Specifying Disciplines for Scopes

The several methods for specifying disciplines allow you to fine-tune the discipline resolution process. This section provides guidance for applying 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 `-disres none`.
Has clearly defined boundaries between analog and digital	Turn off discipline resolution where it is not required by specifying `-setdiscipline "no_dr` *`scope discipline`*`"`. Use `-disres default` or `-disres detailed` 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 `-setdiscipline "`*`scope discipline`*`"`. Use `-disres default` or `-disres detailed` 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 a `prop.cfg` file or a `modelpath` setting, there is no need to specify the discipline because the use of the `prop.cfg` file 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
```

Setting BDR Specifications on Verilog-XL Type Libraries

Block-based discipline resolution (BDR) can be applied to Verilog-XL type libraries by using the `VLIB` and `YLIB` BDR directives. The `VLIB` and `YLIB` BDR directives are used in conjunction with the irun/ncelab `-v` *`<XL_library_file>`* and `-y` *`<XL_library_directory>`* options and 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`	`-setdiscipline "vlib-<XL_library_file>- <discipline>"`
	`-scope_discipline "vlib-<XL_library_file>- <discipline>"`

`YLIB`	`-setdiscipline "ylib-<XL_library_directory>- <discipline>"`
	`-scope_discipline "ylib-<XL_library_directory>- <discipline>"`

where *`<XL_library_file>`* maps to the associated `-v` *`<XL_library_file>`* and *`<XL_library_directory>`* maps to the associated `-y` *`<XL_library_directory>`*.

Wildcards are not supported for this feature.

-use5x4vhdl Option

Specifies that configurations apply to VHDL as well as Verilog-AMS, and that configurations take precedence over VHDL default binding and other searches. However, any configuration rules included in the VHDL source, such as `use` clauses, take precedence over the configuration.

Be aware, that if you plan to use a configuration, your design units must be compiled with the `-USe5x` command-line option. For more information, see "Using a Configuration" on page 46.

-wreal_resolution Option

Specifies the `wreal` resolution function you want the elaborator to use.

Valid Values:

Setting	**Description**
`default`	Default setting
`fourstate`	Verilog 4-state logic resolution algorithm
`sum`	Summation of all drivers
`avg`	Average of all drivers
`min`	Minimum value of all drivers
`max`	Maximum value of all drivers

See “Selecting a wreal Resolution Function” on page 244 for more information.

Example ncelab Command Lines

The following command includes the `-MEssages` option, which prints elaborator messages:

```
ncelab -messages top
```

The following example includes the `-LOGfile` option, which renames the log file from `ncelab.log` to `top_elab.log`:

```
ncelab -messages -logfile top_elab.log top
```

In the following example, the `-ERrormax` option tells the elaborator to abort after 10 errors:

```
ncelab -messages -errormax 10 top
```

The following example uses the `-File` option to include a file called `ncelab.args`, which contains a set of elaborator command-line options:

```
ncelab -file ncelab.args top
```

The following example uses the `-SDF_Cmd_file` option to specify an SDF command file called `dcache_sdf.cmd`. Using this option overrides the automatic SDF annotation to Verilog portions of the design. The command file contains commands that control SDF annotation. For details on SDF annotation, see the "SDF Timing Annotation" chapter, in *Cadence Verilog Simulation User Guide*.

```
ncelab -messages -sdf_cmd_file dcache_sdf.cmd top
```

The following example illustrates how to use the `-modelpath` option to elaborate modules that use primitives:

```
ncelab -modelpath "simp_res.m" 2bit_adder.v
```

The following example illustrates how to use a configuration file. In this example, the configuration file is the first design unit specified on the command line, so the simulation snapshot, by default, goes into the view directory of the configuration. As a result, there is no need to explicitly specify the snapshot directory with the `-SNapshot` option.

```
ncelab -use5x4vhdl myconfig.cfg another.top
```

Using hdl.var Variables with ncelab

`ncelab` uses the following `hdl.var` variables. For additional information, see “hdl.var Variables” in the “Elaboration Command-Line Options” book.

hdl.var Variables Used by ncelab	**Description**
`MODELPATH`	Specifies the SPICE or Spectre source files for the models used in your design.
`NCELABOPTS`	Sets elaborator command-line options.
`LIB_MAP`	Specifies the list of libraries to search, and the order in which to search them, when the elaborator resolves instances.
`VIEW_MAP`	Specifies the list of views to search, and the order in which to search them, when resolving instances.
`WORK`	Specifies the work library. The elaborator uses this variable only for VHDL default binding.

Using the Simulation Front End (SFE) Parser

The AMS Designer simulator uses the simulation front end (SFE) analog parser. With the SFE parser, the Spectre and UltraSim solvers 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.

Important

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" on page 440 so that you can evaluate the differences between the old Spectre parser and the SFE parser carefully. If you have any questions, contact Cadence.

Features of the SFE Parser

The SFE parser provides:

- Native support for Spectre, SPICE, and HSPICE input files, including Spectre netlist compiled functions (NCFs)

  Note: See "Netlist Compiled Functions (NCFs)" in the *Virtuoso Spectre Circuit 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" on page 440 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" on page 439)
- 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" on page 440)
- 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 formats

  The 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`.

Note: 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")
```

Note: 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:

```
ncelab -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

Binding during Elaboration

Binding is the process of selecting which design units are instantiated at each location in the design hierarchy. 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.

You can find information on the following binding mechanisms in "Elaboration Command-Line Options" book.

- The default binding mechanism
- The `ncelab -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 `ncelab -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

  Note: See also "-modelpath Option" on page 423.

These binding mechanisms do not apply to the top-level modules that you specify on the command line. See "Specifying Design Units for Elaboration" on page 418 for information on the rules for selecting a Library.Cell:View for top-level modules.

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 `prop.cfg`:

   ```
   resistor
   capacitor
   inductor
   tline
   vcvs
   vccs
   ```

2. 5x config rules
3. `-modelpath` setting

Enabling Read, Write, or Connectivity 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.

The following topics are discussed in "Enabling Read, Write, or Connectivity Access to Simulation Objects" in the *ElaborationCommand-Line Options* book.

- The limitations imposed by running in regression mode
- How to turn on read, write, and connectivity access by using the following elaborator (`ncelab`) 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. See also "Generating an Access File" for information about how to generate an access file automatically.
- General guidelines for setting access control

Selecting a Delay Mode

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.

For details on selecting and using delay modes, see “Selecting a Delay Mode” in the “Elaborating the Design With ncelab” chapter of the Elaborating Your Design book.

Setting Pulse Controls

In the AMS Designer simulator, 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.

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

For more information, see “Setting Pulse Controls” in the *Elaboration Command-Line Options* book.

15 Simulating

After you have compiled and elaborated your design, you can run the simulator using `ncsim`. This program uses the compiled-code streams to simulate the dynamic behavior of the design.

Note: `irun` runs `ncsim` automatically during the simulation phase.

`ncsim` 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.

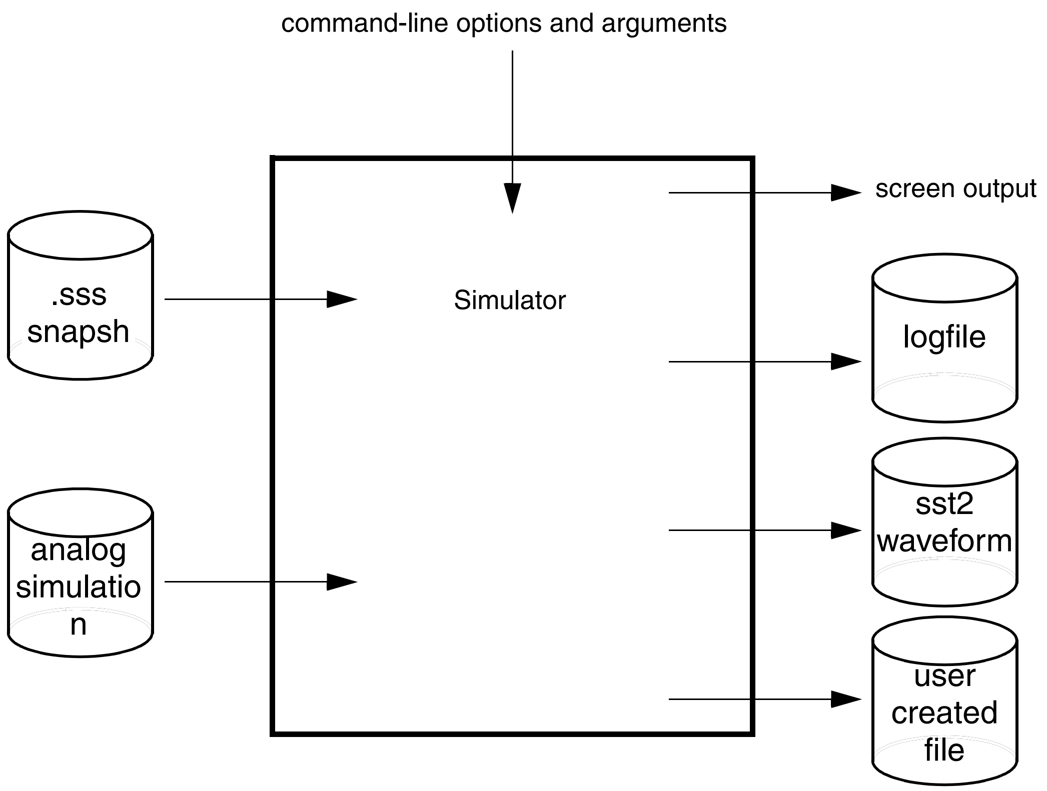
You control the outputs of simulation using the model, the analog simulation control file, or the debugger. Result files can come from the model, Simulation History Manager (SHM) databases, or Value Change Dump (VCD) files.

See the following topics for more information:

- Exiting the Simulation on page 470

Diagram Illustrating Simulator Inputs and Outputs

The following figure illustrates some of the inputs and outputs that the simulator can use and generate.



ncsim Command Syntax and Options

The `ncsim` command has the following syntax:

`ncsim` `[`*`options`*`] [`*`Lib.`*`]`*`Cell`*`[:`*`View`*`]`

For the complete set of `ncsim` command options, see "ncsim Command Options" in the "Simulating Your Design with ncsim" chapter in the *Simulating Your Design* book.

The AMS-specific options are as follows:

Option	Description
`-ANalogcontrol` *`control_file`*	Specifies the analog simulation control file to use. For additional information, see "-ANalogcontrol Option" on page 457.
`-amsformat <sst2|sst2_all|psfxl|psfxl_all>`	

Option	Description
	Controls the storage format for AMS probes. `sst2`: Sets the Tcl-based probes to use sst2 storage. In this mode, the `rawfmt` option specified in the analog control file is honored. This is the default. `sst2_all`: Sets the Tcl probes to use sst2 storage. However, this option also overrides the `rawfmt` option specified in the analog control file so that all analog probes are stored in the sst2 format in the SHM database. In other words, this option overrides the `rawfmt` setting in the analog control file to `rawfmt=sst2`. `psfxl`: Enables the unified PSFXL/SST2 waveform database storage. This mode sets the TCL analog probes to use psfxl storage in the SHM waveform database. The `rawfmt` option in the analog control file is honored. Note that when `rawfmt=psfxl` or `rawfmt=sst2` is specified, the SPICE probes are stored in the default PSFXL/SST2 database in the `psfxl` format. For all other cases, the `rawfmt` data is stored in the specified format in the `.raw` directory. `psfxl_all`: Enables the unified PSFXL/SST2 waveform database storage. This mode sets the Tcl probes to use psfxl storage and also overrides the `rawfmt` option specified in the analog control file so that all analog probes are stored in the psfxl format in the SHM database. In other words, this option overrides the `rawfmt` setting in the analog control file to `rawfmt=psfxl`.

Option	Description

`-aps_args`	Specify one or more command-line arguments for running an AMS Designer simulation using the APS solver. You can also include multiple entries of the `-aps_args` parameter on the command line, which are concatenated during command processing.
	
	Note: The `-aps_args` parameter is ignored if the Spectre or UltraSim solver is selected.
	
	Valid APS solver arguments in AMS include:
	`+errpreset`
	`+lorder`
	`+mt[=`*`N`*`]`
	`+multithread[=`*`N`*`]`
	`-mt`
	`-multithread`
	`+parasitics`
	`+lqtimeout`
	`+lqsleep`
	`+lsuspend`
	`-lsuspend`
	`+rtsf`
	`-cmiconfig`
	`-cmiversion`
	`-h`
	`-plugin` *`plugin_path`*
	`-r`
	`-raw`
	
	See the *Virtuoso Accelerated Parallel Simulator User Guide* for more information about these arguments.
	
	You can achieve parasitics reduction for RF circuits by using the `+parasitics [=`*`N`*` | rf]` argument with the `-aps_args` parameter.
	
	`-aps_args +parasitics=[`*`N`*` | rf] ...`
	
	Where the value specified for the `+parasitics` argument represents the maximum frequency (in GHz) of interest for RF 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`

Option	Description

	■ If no value is specified for the `+parasitics` argument, the maximum frequency of `1 GHz` is applied by default.
	
	Note: If you specify more than one argument, you must separate them with a space and enclose them within quotation marks.
	
	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.
	
	`ncsim -aps_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.
	
	`ncsim -aps_args "+lqs 300"`
	
	The `+lsuspend` argument (applied by default) allows you to suspend or resume the license for APS 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.

`-CDS_IMPLICIT_TMPDir` *`implicitTmpDir`*	
	Directs the simulator to use the snapshot in the *`implicitTmpDir`* directory.

Option	Description
`-MOdelpath "[cell-[`*`lib_name`***`.]`***`cell_name`***`[:`***`view_name`***`]-]`** ***`pathname`*** **`[(`***`section`***`)]{:`** ***`pathname`*** **`[(`***`section`***`)]}"`**	
	Same as `ncelab -modelpath` except that models you specify using `-modelpath` on the `ncsim` command line override any models you specified during `ncelab` using the `-modelpath` option on the `ncelab` command line, or the `MODELPATH` variable in your `hdl.var` file, or in your `prop.cfg` file. For more information, see "-MOdelpath Option" on page 457.
`-SImcompatible_ams` *`compat_val`*	
	For the AMS Designer simulator with the UltraSim solver: Specifies whether to set simulation values for compatibility with the Spectre language or with the HSPICE language. The default is `hspice`. For more information, see "-SImcompatible_ams Option" on page 458.
`-solver` `spectre | aps`	Specify whether the Spectre solver or the APS solver is to be used with the AMS Designer simulator. **Note:** `-solver ultrasim` is not a valid argument with the `ncsim` command. When you are using the three-step approach to run a design, the only way to select the UltraSim solver is to use the `ncelab -amsfastspice` command while elaborating the design.

Option	**Description**

`-spectre_args`	Specify one or more Spectre command-line arguments. You can also include multiple entries of the `-spectre_args` parameter on the command line, which are concatenated during command processing.
	
	Note: The `-spectre_args` parameter is ignored if the APS or UltraSim solver is selected.
	
	Valid Spectre arguments in AMS include:
	
	`+lorder`
	`+mt[=N]`
	`+multithread[=N]`
	`-mt`
	`-multithread`
	`+parasitics`
	`+aps`
	`+rtsf`
	`+lqtimeout`
	`+lqsleep`
	`+lsuspend`
	`-lsuspend`
	`-cmiconfig`
	`-cmiversion`
	`-h`
	`-plugin` *`plugin_path`*
	`-r`
	`-raw`
	`-V`
	`-W`
	
	See the *Virtuoso Spectre Circuit Simulator User Guide* and the *Virtuoso Spectre Circuit Simulator Reference* for information about these arguments.
	
	You can achieve parasitics reduction for RF circuits by using the `+parasitics [=`*`N`* | rf`]` argument with the `-spectre_args` parameter.
	
	`-spectre_args +parasitics=[`*`N`*` | rf] ...`
	
	Where the value specified for the `+parasitics` argument represents the maximum frequency (in GHz) of interest for RF 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`

Option **Description**

- If no value is specified for the `+parasitics` argument, the maximum frequency of `1 GHz` is applied by default.

Note: If you specify more than one argument, you must separate them with a space and enclose them within quotation marks like this:

```
    ncsim -spectre_args "-raw ../psf
```

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.

```
ncsim -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.

```
ncsim -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.

Note: Stand-alone Spectre does not have `+lsuspend` by default; this is an AMS-only behavior. This command will have an effect only on the analog licenses.

Option	Description

`-ultrasim_args`	Specify one or more UltraSim command-line arguments. You can also include multiple entries of the `-ultrasim_args` parameter on the command line, which are concatenated during command processing.
	
	Note: The `-ultrasim_args` parameter is ignored if the APS or Spectre solver is selected.
	
	Valid UltraSim arguments in AMS include:
	
	`+lorder`
	`+rtsf`
	`-turbo`
	`-plugin` *`plugin_path`*
	
	The `-turbo` argument is used to turn off the UltraSim-Turbo feature, which is available only in `sim_mode=a` and is turned on by default in this mode.
	
	The `+rtsf` argument enables RTSF, which is a PSF extension that can plot extremely large datasets (where signals have a large number of data points, for example 10 million) within seconds.
	
	Note: If you specify more than one argument, you must separate them with a space and enclose them within quotation marks.

`-UPdate`	Recompiles out-of-date design units as necessary. Notice that the behavior of the `-UPDate` option is affected when the `-CDS_IMPLICIT_TMPOnly` option is stored in the snapshot header.
`-uselicense` *`mnemonic_list`*	Specifies a prioritized list of colon-delimited mnemonics to select the license for simulation. See "-uselicense Option" on page 459 for more information.

The *`Lib`*, *`Cell`*, and *`View`* together identify the snapshot. You can simulate using a snapshot that has a time stamp that is earlier than the latest snapshot for a cell (see "Restarting the Simulator from a Previously-Saved Snapshot" on page 464. (Specifying the snapshot also automatically specifies the SSI, which has the same name.) You can specify the options and the snapshot argument in any order, provided that the parameters to an option immediately follow the option.

- If a snapshot with the same name exists in more than one library, Cadence recommends you specify both the cell and the library.
- If there are multiple views that contain snapshots, Cadence recommends you specify both the cell and the view.

If you do not specify a library or a view, the simulator uses a set of rules to resolve the snapshot reference on the command line. For more information, see "Rules for Resolving the Snapshot Reference" in the "Simulating Your Design with ncsim" chapter of the *Simulating Your Design* book.

You can specify `ncsim` command options in upper or lower case and abbreviate them to the shortest unique string. For example, you can specify just `-an` for `-analogcontrol`. In the following sections, the shortest unique string is indicated with capital letters.

-ANalogcontrol Option

Specifies the analog simulation control file to use. The analog simulation control file is an ASCII text file written in the Spectre, or Spectre and UltraSim, control languages. The contents of the file control the behavior of the analog solvers. For example,

```
ncsim top:amsSS -cdslib /SAR_A2D/cds.lib -hdlvar /SAR_A2D/hdl.var -analogcontrol
/SAR_A2D/tutorial_run/amsSC.scs
```

For detailed information about the analog simulation control file, see Chapter 4, "Specifying Controls for the Analog Solvers."

-MOdelpath Option

You can use the `-modelpath` option on the `ncsim` command line to override any models you specified using `-modelpath` on the `ncelab` command line, or the `MODELPATH` variable in your `hdl.var` file, or in your `prop.cfg` file. You can use `ncsim -modelpath` to change model files using the save-and-restart feature. You must observe the following rules:

- You cannot introduce new scope during simulation: All scopes must be same as those defined during `ncelab`.
- You can specify new model files for any scopes you defined during `ncelab`.
- You can introduce new non-scoped model files during simulation.

Caution

If you introduce any topology changes, you might observe strange behavior or core dumps.

-SImcompatible_ams Option

For the AMS Designer simulator with the UltraSim solver:
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:

```
ncsim -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.

See also "Switching Languages in the Analog Simulation Control File" on page 54.

-uselicense Option

The -`uselicense` option lets you specify a custom prioritized license checkout order for simulation. For example:

```
-uselicense mnemonicList[:DEFAULT]
```

where *`mnemonicList`* is a colon-separated list of one or more valid mnemonic keywords. The optional `DEFAULT` mnemonic indicates the default license selection, which includes digital-specific license mnemonics such as `NCVLOG` and `NCSIM`.

For more information about these and other digital-specific mnemonics, see the Product and Licensing Information chapter. Also see `-uselicense` in the "Simulating Your Design with ncsim" chapter of the *Simulating Your Design* book.

AMS Designer Verification Option

More than 80% of today's integrated circuit have analog content, the majority of which is tightly coupled with digital. The established digital flows cannot handle the fast growing mixed-signal SoC design starts, while existing mixed-signal technology is limited to small isolated blocks.

The new AMS Designer Verification option provides a smooth mixed-signal verification flow that extend today's mature digital SoC verification methodologies to analog content/ mixed-signal content. It provides a complete solution for mixed-signal SoC verification by enhancing the performance and capacity of existing AMS block-level technology, supporting cross domain connectivity between test benches and IP from multiple vendors.

The advanced features required for SoC Verification that the AMS Designer Verification option provides are as mentioned below. These features require an AMS Designer Verification option license.

- SV Real-to-Electrical connection - more detailed information about this feature will be provided in the future revision of the documentation.
- Electrical-to-SV Real Connection - more detailed information about this feature will be provided in the future revision of the documentation.
- VHDL-SPICE - refer to Connecting VHDL Blocks to SPICE Blocks.
- AMS-CPF (CPF enabled AMS with power-smart connection modules) - refer to Using Common Power Format with AMS Designer.

Example ncsim Command Lines

The following command runs the simulator in noninteractive mode. This command automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input. The `ncsim` program reads controls for the analog solver from `mycontrolfile`.

```
ncsim -input setup.inp -analogcontrol mycontrolfile top
```

The following command runs the simulator in noninteractive mode with the SimVision environment. This command automatically starts the simulation or the processing of commands from `-input` options without prompting you for command input.

```
ncsim -input setup.inp -gui -run top
```

The following command runs the simulator in interactive mode. The simulation waits at time 0 for interactive input.

```
ncsim -tcl top
```

The following command runs the simulator in interactive mode with the SimVision environment. The simulation waits at time 0 for interactive input.

```
ncsim -gui top
```

The following command uses the `-logfile` option to rename the log file from the default (`ncsim.log`) to `top.log`.

```
ncsim -messages -run -logfile top.log top
```

The following command uses `-errormax 10` to tell the simulator to stop after 10 errors.

```
ncsim -errormax 10 top
```

The following command uses the `-file` option to include a file called `top.vc`, which includes a set of command line options, such as `-messages`, `-nocopyright`, `-logfile`, and `-errormax`.

```
ncsim -file top.vc top
```

The following command uses the `-input` option to source the file `top.inp` at initialization. This file contains a sequence of simulator (Tcl) commands.

```
ncsim -input top.inp top
```

The following command uses the `-keyfile` option to specify that the name of the key file is `top.key` instead of the default `ncsim.key`. You could use this key file to reproduce an interactive session by using the file name `top.key` as the argument to the `-input` option.

```
ncsim -tcl -keyfile top.key top
```

The following command runs a simulation on the AMS Designer simulator using the APS solver.

```
ncsim -solver aps top
```

The following set of commands performs all the steps necessary to prepare and run a simulation that uses the UltraSim simulator.

```
ncvlog -ams test.v
ncelab -amsfastspice -propspath prop.cfg test
ncsim -analogcontrol test.scs test
```

The `ncvlog` step is the same as that used for the Spectre solver. The `ncelab` step uses `-amsfastspice` to configure the elaboration for the UltraSim solver. The `-propspath` option points to a file that specifies where the source files are for Spectre and SPICE design blocks. The elaborator includes `-amsfastspice` and `-propspath` information in the snapshot that it produces so you do not need to specify them on the `ncsim` command.

hdl.var Variables

If you are running the AMS Designer simulator using the three-step approach, the simulator recognizes and uses the following `hdl.var` variables:

- `NCSIMOPTS`

  Sets simulator command-line options. You can include a snapshot name. For example:

  ```
  DEFINE NCSIMOPTS -messages
  ```

- `WORK`

  Specifies the default library for the snapshot. If the snapshot is not found in this library, the rest of the libraries in the `cds.lib` file are searched.

Running the Simulator

You can run the `ncsim` simulator in two modes:

- 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 `ncsim>` 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.

  See "Controlling the Simulation" in the *Running SimVision in Simulation Mode* for more information.

Restarting the Simulator from a Previously-Saved Snapshot

Using the three-step method, you compile your modules using `ncvlog`, then you elaborate using `ncelab` which elaborates only newly-compiled modules: If you have not recompiled a module, `ncelab` does not re-elaborate it. The third step is to simulate using `ncsim`.

When you simulate using a previously-saved snapshot, the simulator (`ncsim`) 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

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 `ncsim` using snapshot `B`.

   If you run `ncsim` using snapshot `A` at this point, you cannot use Tcl `restart` to simulate using snapshot `B` because snapshot `A` now has a newer time stamp than snapshot `B`. You can simulate snapshot `B` only from the `ncsim` command line.

Use Model 2

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 `ncsim` using snapshot `B`.
   You can run `ncsim` using snapshot `A`.
   You can also use Tcl `restart` to simulate using snapshot `B`.

However, if you run `ncsim` 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 `ncsim` command line.

Updating Design Changes When You Run the Simulator

After editing a design unit, you can use `ncsim -update` to recompile and re-elaborate all out-of-date design units before re-simulating. For more information, see "Updating Design Changes When You Invoke the Simulator" in the *Simulating Your Design* book.

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), see "Providing Interactive Commands from a File" in the *Simulating Your Design* book.

Using the Save-and-Restart Feature

Using the AMS Designer simulator with the simulation front end (SFE) parser, you can save a snapshot of a simulation and restart after making changes to simulation parameters and models. See the following topics for more information:

- Stopping the Simulation and Saving the Current Simulation State
- Making Changes and Restarting the Simulator on page 468

See also *Simulating Your Design* and *Debugging Your Design*.

If you are using `irun`, see "Using the Save-and-Restart Feature of the AMS Designer Simulator" in the *Virtuoso® AMS Designer Simulator Tutorials* book.

Important

> If you want to save and restart using the Spectre solver or the APS solver, you must be using the SFE parser. You can use save-and-restart either in non-interactive mode or in Tcl mode.
>
> When running AMS-UltraSim, you can save a snapshot either in non-interactive (command-line) mode or in Tcl mode, and you can restart either from command-line mode or Tcl mode.

You can also use the save-and-restart feature in AMS 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 AMS 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.

Note: The save-and-restart feature for AMS designs with SystemC does not work on SLES11.

Stopping the Simulation and Saving the Current Simulation State

Here are some examples of how you can stop the simulation:

- Use the Tcl `run` command and specify a stop time.
- Type *Ctrl+C* to interrupt the simulation.

- In SimVision, click the pause button to interrupt the simulation.

Use the Tcl save command to create a snapshot of the current simulation state. For example:

```
ncsim> save top:savedSnapshot
```

Making Changes and Restarting the Simulator

Before you restart the simulator (by specifying a saved snapshot on the `ncsim` command line) you can make the following design changes:

- You can change simulation parameters that do not affect the circuit topology such as `reltol`, `abstol`, `errpreset`, `method`, and `stop`. Use `ncsim -analogcontrol` to specify the analog control file containing the changed parameters. If you are using the Spectre solver, you can also change the `reltol`, `abstol`, and `stop` parameters using the Tcl `analog` command. If you change any of these options using both methods, the Tcl `analog` command takes precedence; however, Tcl changes are not part of the saved snapshot information.

- You can change analog SPICE or Spectre models for the simulation by editing or replacing the model files whose paths you specified using the `ncelab -modelpath` option; or you can change the path to or name of the model file you want the simulator to use using the `ncsim -modelpath` option (unless you are using the AMS Designer simulator with the UltraSim solver and the UltraSim front end parser, which does not support using the `ncsim -modelpath` option).

  If you are using the Spectre solver, you can change model parameters in Spectre and SPICE and analog control files, but you cannot change the circuit topology: You must not replace subcircuit definitions or Verilog-A modules such that you remove or introduce nodes or use different devices in your design with respect to the original Spectre/SPICE files you submitted to `ncsim` for simulation.

Caution

The AMS Designer simulator with the Spectre solver does not detect circuit topology changes automatically; such changes can result in unexpected behavior, including crashes.

If you are using the UltraSim solver, you can change the circuit topology (*but Cadence strongly discourages it*). When you restart, the software initializes any added nodes as if the operating point was not saved and ignores references to any deleted nodes. The software initializes the rest of the nodes to their previously saved simulation values.

So, if you are changing simulation parameters and using a different model file when you restart the simulation, your `ncsim` command line might look something like this:

```
ncsim top:savedSnapshot … -analogcontrol /myProject/myRun/changedParams.scs …
-modelpath "differentModels.m"
```

Regarding scoping and model file changes:

- You must not introduce any new scopes during simulation: All scopes must be the same as those defined during elaboration.

  See "-modelpath Option" on page 423 for information about model files and scoping defined during elaboration.

- You can specify new model files for scopes defined during elaboration.

  See "-modelpath Option" on page 423 for information about model files and scoping defined during elaboration.

- You can introduce new non-scoped model files during simulation.

Switching SPICE Blocks from an Existing Snapshot

To reduce simulation time to a large extent, you can switch the SPICE blocks in your design with other equivalent SPICE blocks from an existing snapshot. This is especially useful for large designs that take a long time to simulate.

The AMS Designer simulator allows you to run the simulation using one representation of a SPICE block in your design, save the results, and then restart the simulation using another representation of the same SPICE block.

Example

The simulation can be restarted using a specific snapshot name, say `s1`, using the following command:

```
irun -r :s1 <NEW_SPICE_DECK.scs>
```

where `NEW_SPICE_DECK.scs` refers to the new SPICE content that will replace its existing counterparts in snapshot `s1`.

Alternatively, you can restart the simulation using the last generated snapshot by using the following command:

```
irun -R <NEW_SPICE_DECK.scs>
```

where `NEW_SPICE_DECK.scs` refers to the new SPICE content that will replace its existing counterparts in the last snapshot (`s1`, in this example).

The following considerations must be kept in mind while using this save/restart solution:

- The Verilog-SPICE boundary must be preserved during the save/restart. No changes in port order and port name are allowed.

- You must provide only one top level SPICE file (deck) during restart on the irun command-line.
- The new SPICE deck can only bring in changes related to the SPICE-to-SPICE domain.
- The SPICE deck should be self-sufficient in itself and should support simulation.
- This solution is supported only in the command-line flow. It is not supported in the ADE flow.

Exiting the Simulation

➤ Do one of the following to exit the simulator,

- ❑ If you are using the Tcl command-line interface, type `exit` or `finish`.

  The `exit` command is a built-in Tcl command. It halts execution and returns control to the operating system. For details, see "exit" in *Incisive Simulator Tcl Command Reference*.

  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`.

  Note: If you type `finish`, the window disappears before you can read the information. However, the information also appears in the log file.

16

Using SimVision with the AMS Simulator

When you use the SimVision environment with the Virtuoso® 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 `irun` or `ncsim` 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 `irun` or `ncsim` command).

See the following topics for more information about using SimVision with the AMS simulator:

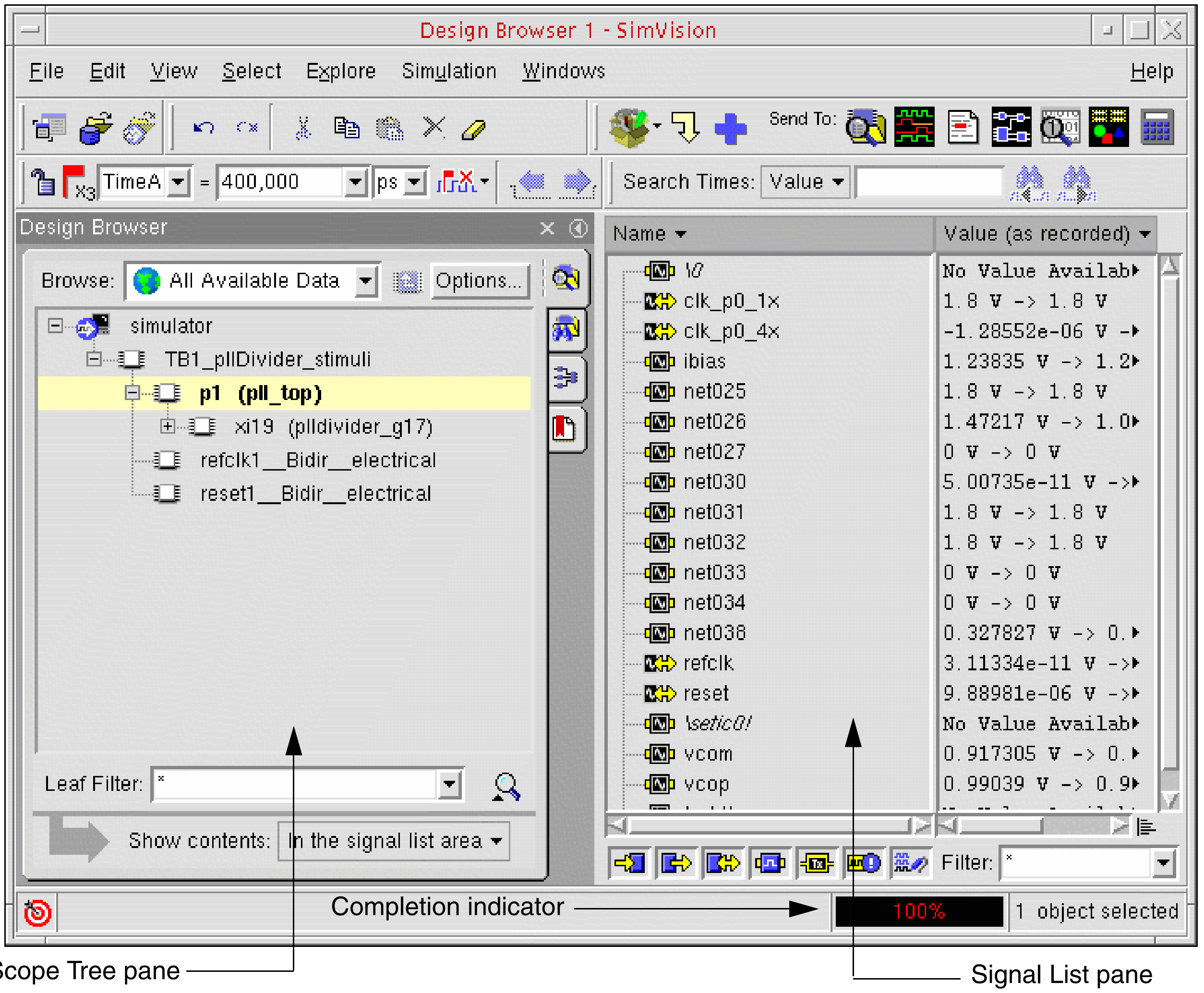
- The Design Browser Window for AMS Designs on page 472
- The Console Window on page 483
- Cross-Probing Instances and Nets on page 484

See also "Viewing Analog Data in the Waveform Viewer" in the *Using the Waveform Viewer* book.

Note: For information on analyzing and debugging purely digital designs, see the *Running the SimVision Analysis Environment* book.

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:



Name	Value (as recorded)
\0	No Value Availab▸
clk_p0_1x	1.8 V -> 1.8 V
clk_p0_4x	-1.28552e-06 V -▸
ibias	1.23835 V -> 1.2▸
net025	1.8 V -> 1.8 V
net026	1.47217 V -> 1.0▸
net027	0 V -> 0 V
net030	5.00735e-11 V ->▸
net031	1.8 V -> 1.8 V
net032	1.8 V -> 1.8 V
net033	0 V -> 0 V
net034	0 V -> 0 V
net038	0.327827 V -> 0.▸
refclk	3.11334e-11 V ->▸
reset	9.88981e-06 V ->▸
\setic0!	No Value Availab▸
vcom	0.917305 V -> 0.▸
vcop	0.99039 V -> 0.9▸

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.

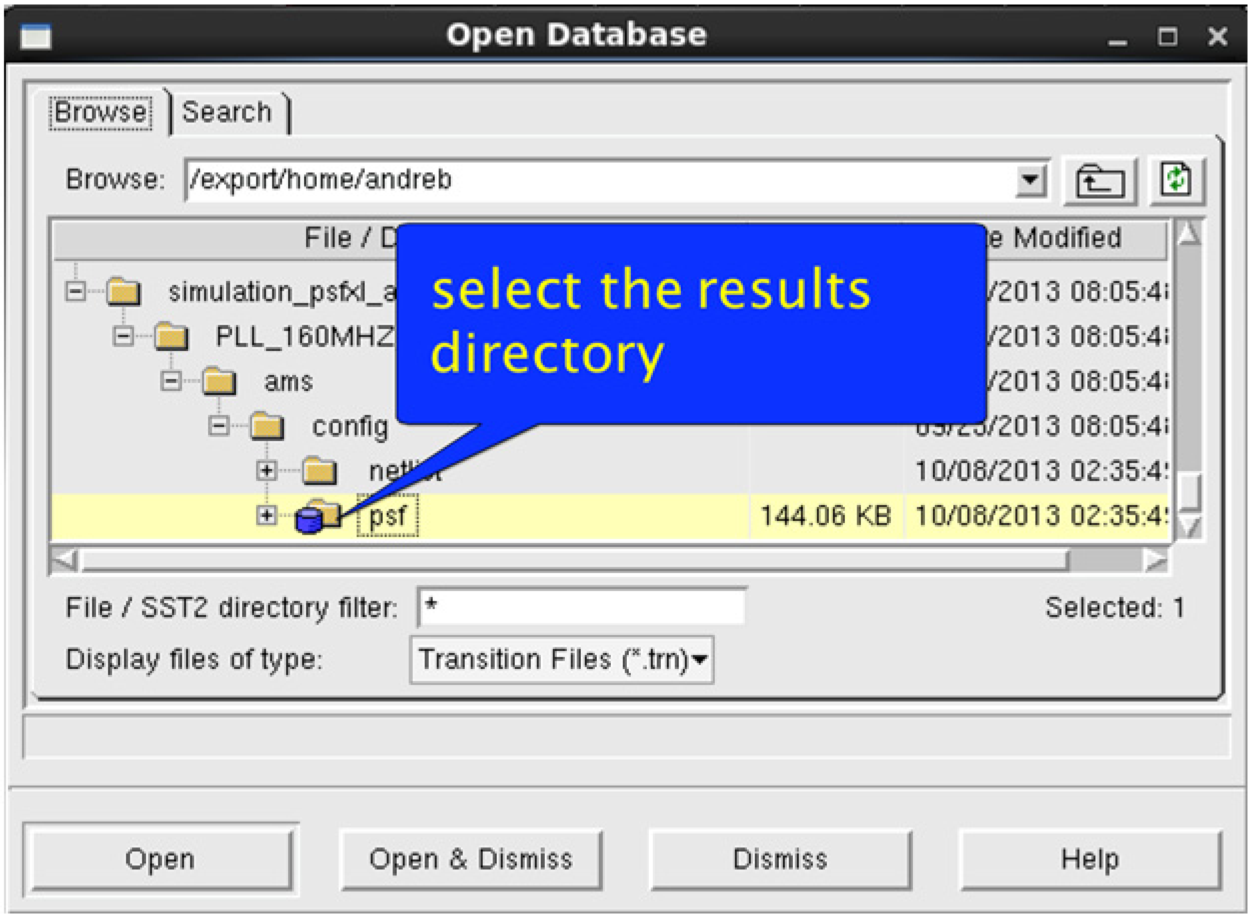
The SimVision environment can also read PSFXL analog simulation data. It supports a unified PSFXL/SST2 database that contains both SST2 digital data and PSFXL analog data, so that you do not need to manually segregate analog probes in a separate database when probing or opening databases to view the results.

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.

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.



Browse: /export/home/andreb

File / D		e Modified
simulation_psfxl_a		/2013 08:05:4
PLL_160MHZ		/2013 08:05:4
ams		/2013 08:05:4
config		/2013 08:05:4
net		10/08/2013 02:35:4
psf	144.06 KB	10/08/2013 02:35:4

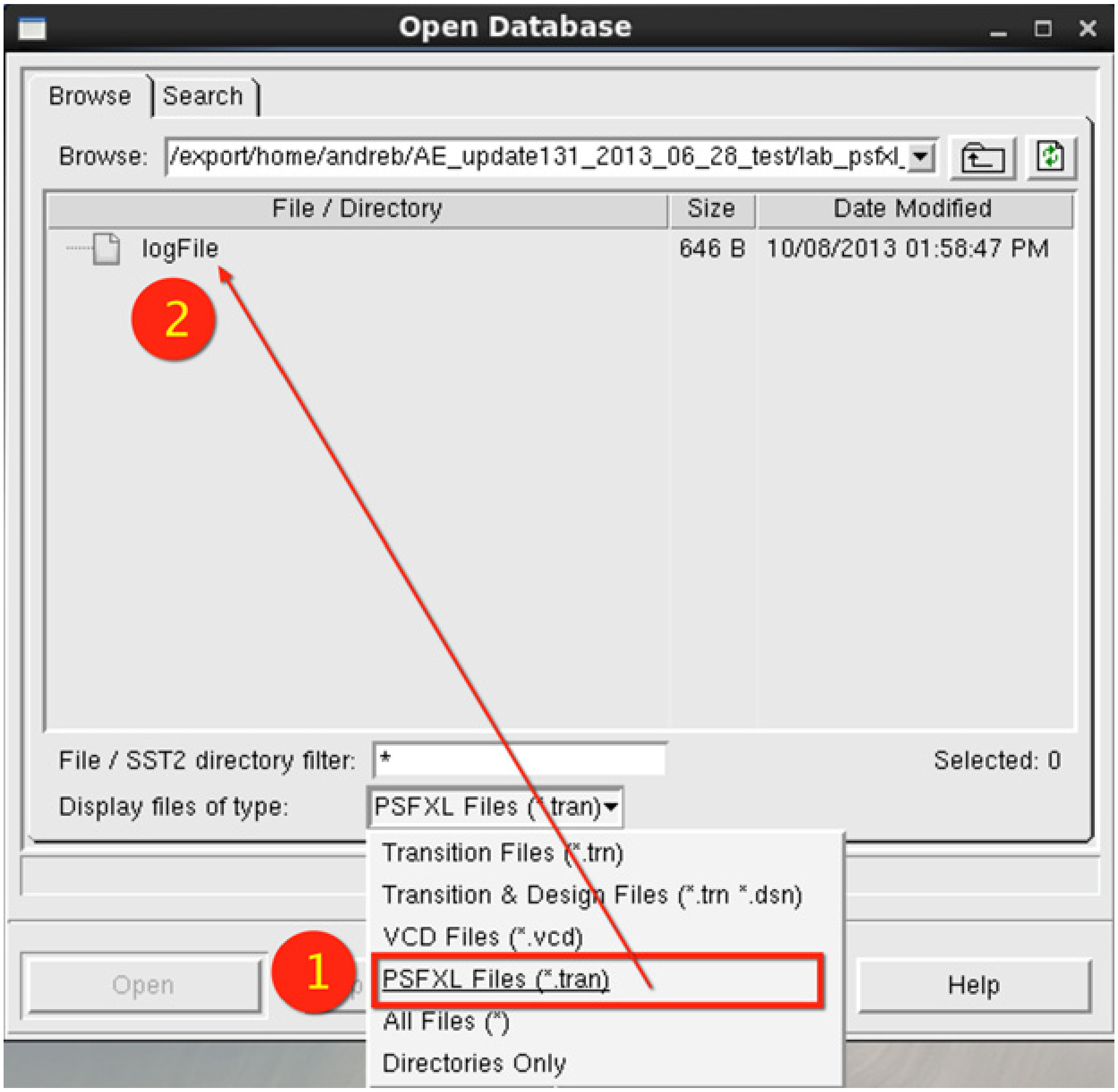
File / SST2 directory filter: *
Selected: 1
Display files of type: Transition Files (*.trn)

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*.



Browse: /export/home/andreb/AE_update131_2013_06_28_test/lab_psfxl_

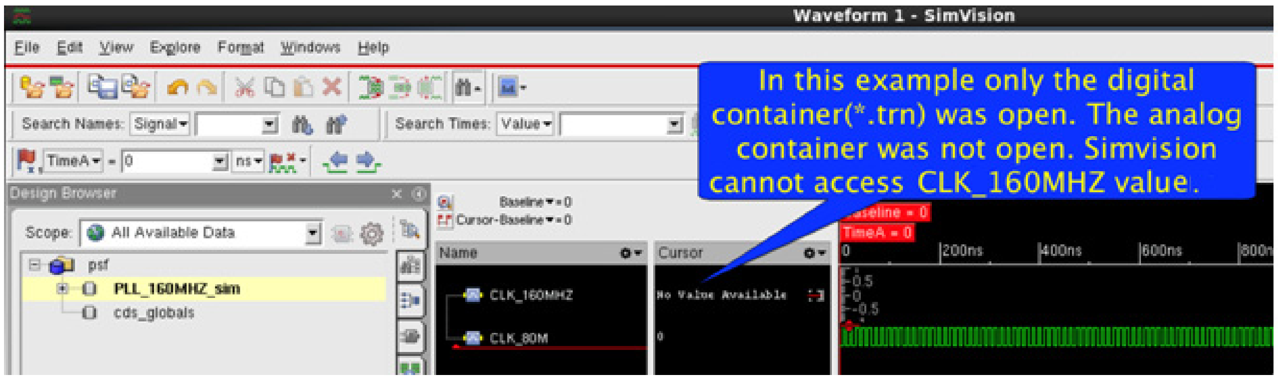
File / Directory	Size	Date Modified
logFile	646 B	10/08/2013 01:58:47 PM

File / SST2 directory filter: *

Display files of type: PSFXL Files (*.tran)

The unified database containing only the analog container is displayed in SimVision.

Note: 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.



You need to specify the `-amsformat <psfxl | psfxl_all>` ncsim or irun option to enable PSFXL/SST2 storage for analog probes.

See the following topics for more information:

- Using the Menus and Forms for AMS Designs on page 476
- Setting Display and Formatting Preferences for Verilog-AMS Objects on page 477
- Selecting Objects on page 480
- Finding Edges on page 480
- Using the Source Browser on page 481
- Editing Source Information on page 482
- Plotting Signals in the Waveform Window on page 482

Using the Menus and Forms for AMS Designs

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. (For information about other SimVision menu selections, see the *Introduction to SimVision* book.)

SimVision Window Menu Choices

Menu item	Comments and cross-references
Edit – Preferences	See "Setting Display and Formatting Preferences for Verilog-AMS Objects" on page 477.
Select – Branches	See "Selecting Objects" on page 480.
Simulation – Advance – To Synchronization Point	Run 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 *Simulation – Create Force*, are available only when the digital solver is active.) **Note:** The simulator stops at any breakpoint it encounters before it reaches the synchronization point.
Simulation – Advance – Timepoint	Simulate up to the beginning of the next analog timepoint or to the next time at which a digital event is scheduled. **Note:** The simulator stops at any breakpoint it encounters before it reaches the specified timepoint.
Simulation – Reset to Start	Not available for mixed-signal designs.
Simulation – Set Breakpoint – Time	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. For more information on setting time points, see "Setting and Managing Breakpoints," in Chapter 8, "Setting and Managing Breakpoints," in the *SimVision User Guide*.

SimVision Window Menu Choices, *continued*

Menu item	Comments and cross-references
Simulation – Deposit Value	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. For more information about using *Deposit Value* for digital quantities, see Chapter 9, “Changing and Monitoring the Value of an Object During Simulation,” in the Cadence *SimVision User Guide*.
Simulation – Create Probe	Probe values of digital and most analog objects to a database. **Note:** To probe the values of currents and Spectre primitives, you must use Tcl `probe` commands. For more information about using *Create Probe* for digital quantities, see Chapter 7, “Creating and Managing Probes,” in the Cadence *SimVision User Guide*.
Simulation – Create Force	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. For more information about using *Create Force* for digital quantities, see Chapter 9, “Changing and Monitoring the Value of an Object During Simulation,” in the Cadence *SimVision User Guide*.

Setting Display and Formatting Preferences for Verilog-AMS Objects

Items available on the Signal List pane for the AMS Designer simulator include analog branch objects.

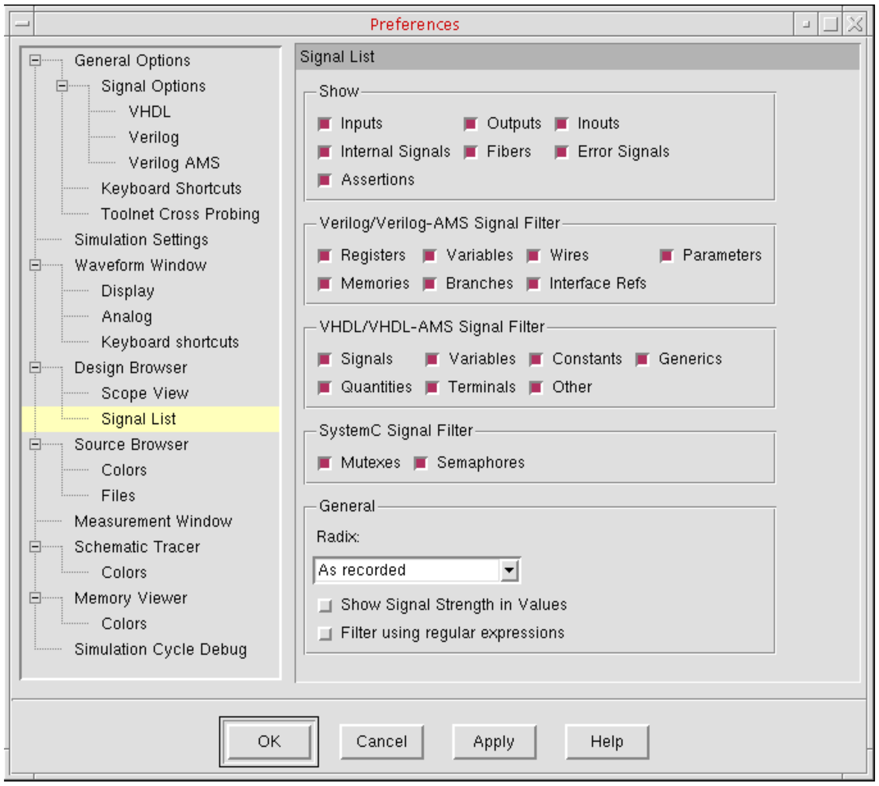
To specify signal list preferences, do the following:

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.



Signal List

Show
- [✓] Inputs
- [✓] Outputs
- [✓] Inouts
- [✓] Internal Signals
- [✓] Fibers
- [✓] Error Signals
- [✓] Assertions

Verilog/Verilog-AMS Signal Filter
- [✓] Registers
- [✓] Variables
- [✓] Wires
- [✓] Parameters
- [✓] Memories
- [✓] Branches
- [✓] Interface Refs

VHDL/VHDL-AMS Signal Filter
- [✓] Signals
- [✓] Variables
- [✓] Constants
- [✓] Generics
- [✓] Quantities
- [✓] Terminals
- [✓] Other

SystemC Signal Filter
- [✓] Mutexes
- [✓] Semaphores

General
Radix: As recorded
- [ ] Show Signal Strength in Values
- [ ] Filter using regular expressions

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.

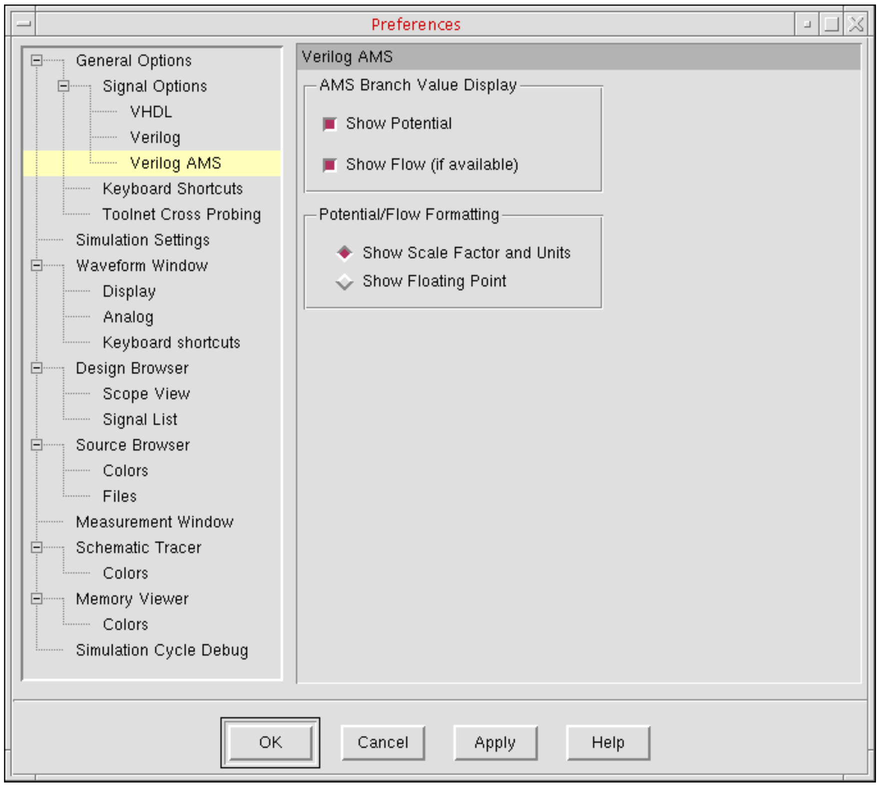
To specify formatting options for Verilog-AMS branches, do the following:

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.



Verilog AMS

AMS Branch Value Display
- [✓] Show Potential
- [✓] Show Flow (if available)

Potential/Flow Formatting
- [✓ Show Scale Factor and Units]
- Show Floating Point

3. In the *AMS Branch Value Display* group box, select the options you want:

Show Potential	Show potential quantities, such as `5.44V`
Show Flow	Show flow quantities, such as `3.45mA`

4. In the *Potential/Flow Formatting* group box, select one of the following format options:

Show Scale Factor and Units	Use scalar multipliers and nature units; for example: `result= -30.646mV`
Show Floating Point	Use floating-point format; for example: `result= -0.0306461`

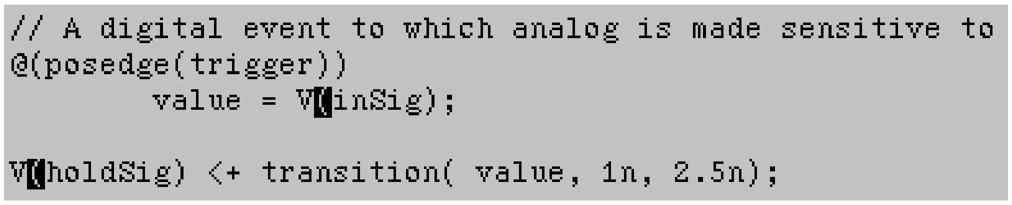
5. Click *Apply*.

Selecting Objects

To select branches in addition to the objects available for a Verilog module, do the following:

➤ In the Design Browser, choose *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);
```

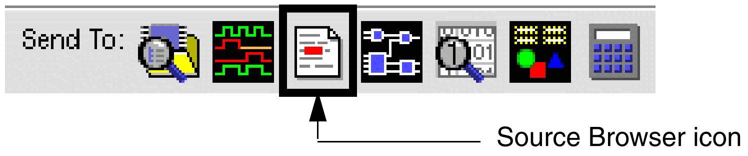
Finding Edges

The *Previous Edge* and *Next Edge* buttons apply to only digital signals, not to analog ones.

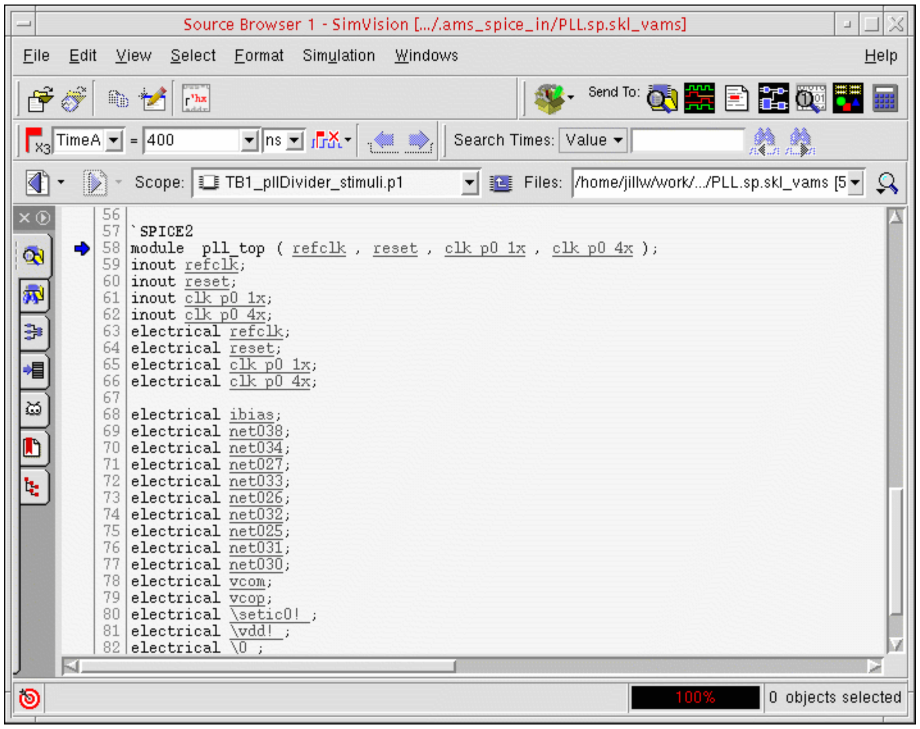
Using the Source Browser

To open the Source Browser, do the following:

➤ In the Design Browser window, click the Source Browser icon.



The text view of the components in your design appears in the Source Browser window.


Source Browser 1 - SimVision [.../.ams_spice_in/PLL.sp.skl_vams]
File
Edit
View
Select
Format
Simulation
Windows
Help
Send To:
TimeA
= 400
ns
Search Times:
Value
Scope:
TB1_pllDivider_stimuli.p1
Files:
/home/jillw/work/.../PLL.sp.skl_vams [5
100%
0 objects selected

```
`SPICE2
module  pll_top ( refclk , reset , clk_p0_1x , clk_p0_4x );
inout refclk;
inout reset;
inout clk_p0_1x;
inout clk_p0_4x;
electrical refclk;
electrical reset;
electrical clk_p0_1x;
electrical clk_p0_4x;

electrical ibias;
electrical net038;
electrical net034;
electrical net027;
electrical net033;
electrical net026;
electrical net032;
electrical net025;
electrical net031;
electrical net030;
electrical vcom;
electrical vcop;
electrical \setic0! ;
electrical \vdd! ;
electrical \0 ;
```

Editing Source Information

You can edit source information, whether the source is text or a schematic, from within the Source Browser.

To edit source information,

1. In the Source Browser, navigate to the module of interest.
2. 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.

Plotting Signals in the Waveform Window

How you plot a signal depends on the kind of probing that you use.

- If you use Tcl probing for analog and digital objects,

  a. Select the net in either the Schematic Editing window, or the Design Browser.

  b. 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,

  a. Start, but do not finish, the simulation.

     For example, you might choose *Simulation – Advance – Timepoint* in the Design Browser.

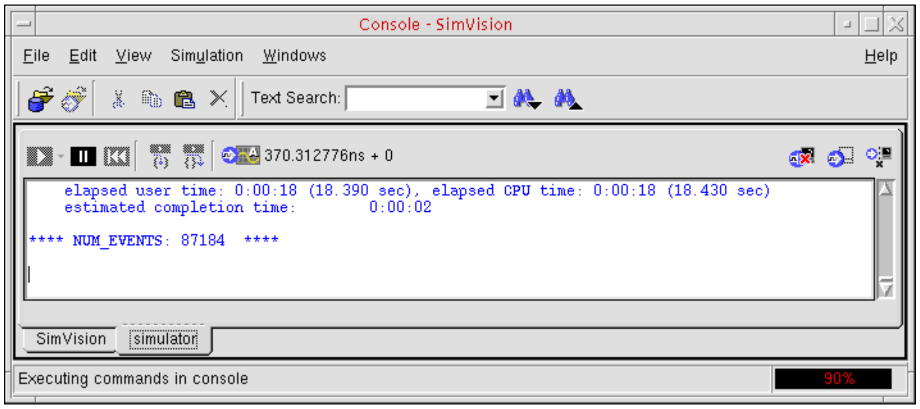
  b. Select the net in either the Schematic Editing window or Design Browser.

  c. 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.

  To plot signals after the simulation finishes, follow the procedures described in Chapter 4, “Accessing the Design Hierarchy,” and Chapter 6, “Managing Simulation Databases,” in the *SimVision User Guide*.

The Console Window

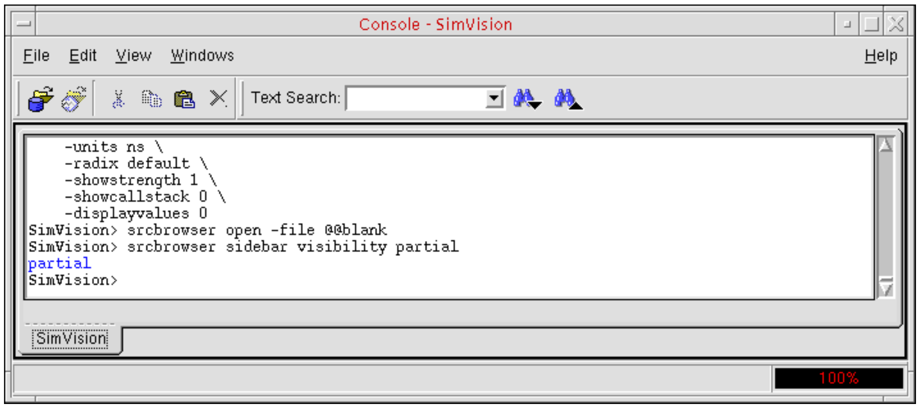
The SimVision Console window gives you access to SimVision and the simulator.



```
    elapsed user time: 0:00:18 (18.390 sec), elapsed CPU time: 0:00:18 (18.430 sec)
    estimated completion time:        0:00:02

**** NUM_EVENTS: 87184  ****
```

Executing commands in console

You can type simulator, SimVision, or Tcl commands on the *SimVision* tab.



```
    -units ns \
    -radix default \
    -showstrength 1 \
    -showcallstack 0 \
    -displayvalues 0
SimVision> srcbrowser open -file @@blank
SimVision> srcbrowser sidebar visibility partial
partial
SimVision>
```

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 on page 485
- Cross-Probing Nets on page 487

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:

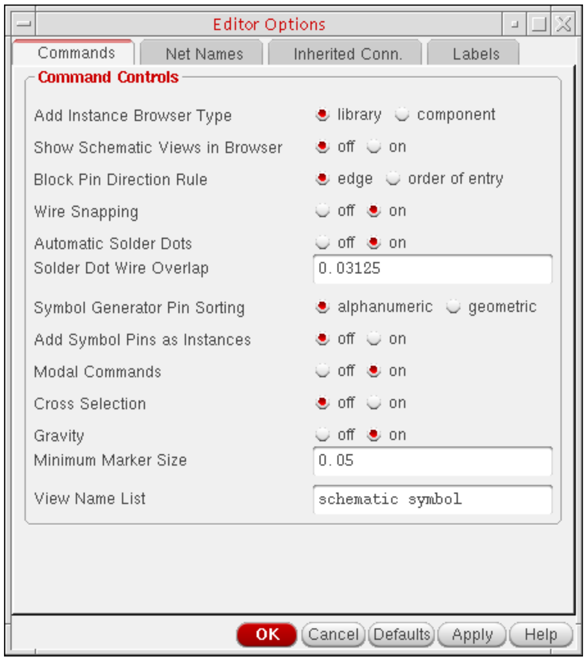
1. In the Cadence hierarchy editor window, choose *View – Tree*.

   Instances in the design appear on the *Tree View* tab.

   Note: 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.



Add Instance Browser Type	[✓ library] component
Show Schematic Views in Browser	[✓ off] on
Block Pin Direction Rule	[✓ edge] order of entry
Wire Snapping	off [✓ on]
Automatic Solder Dots	off [✓ on]
Solder Dot Wire Overlap	0.03125
Symbol Generator Pin Sorting	[✓ alphanumeric] geometric
Add Symbol Pins as Instances	[✓ off] on
Modal Commands	off [✓ on]
Cross Selection	[✓ off] on
Gravity	off [✓ on]
Minimum Marker Size	0.05
View Name List	schematic symbol

3. Turn on *Cross Selection*.

4. 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$_m$	

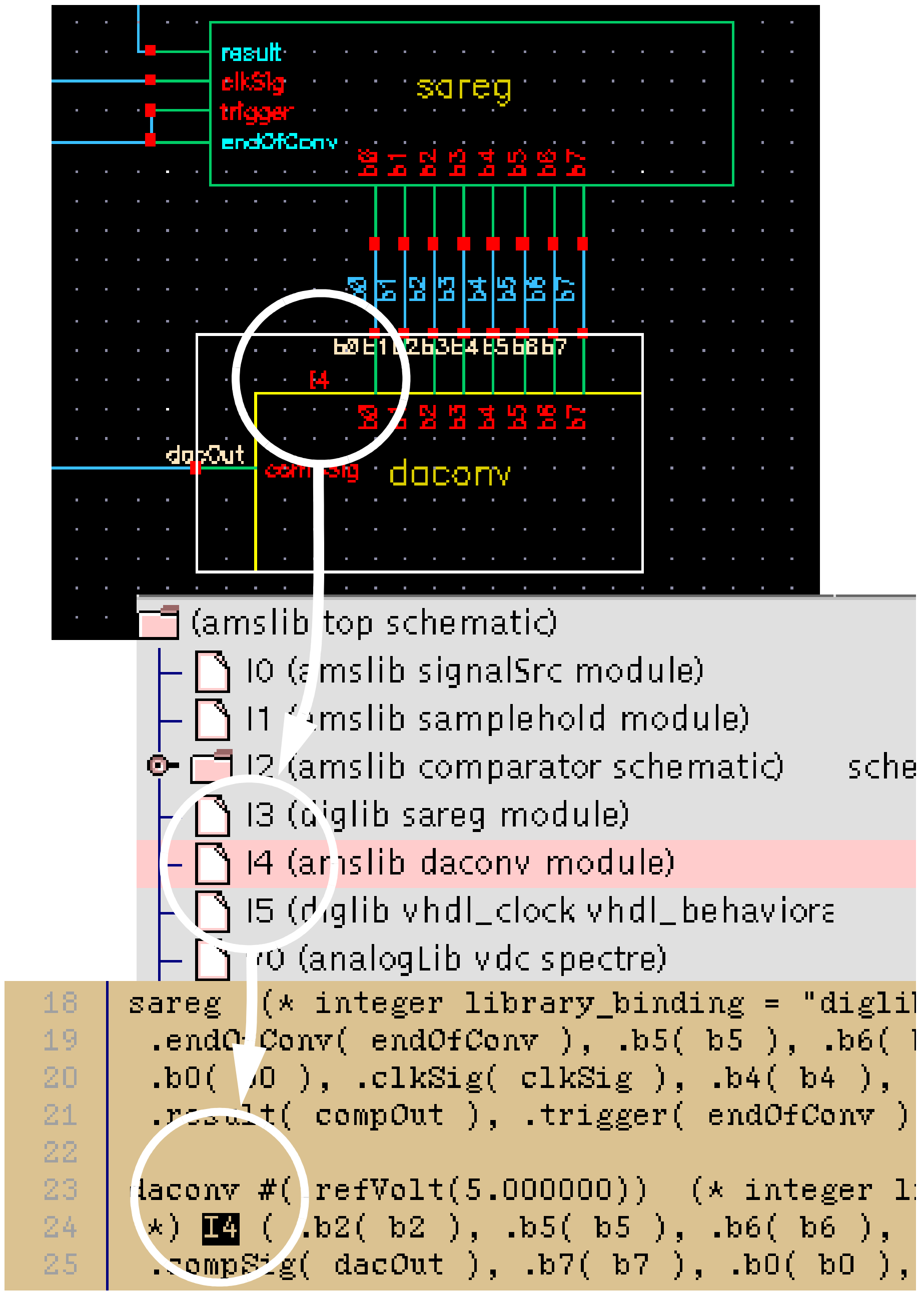
For example, if you select an instance in the schematic editor, the same instance is automatically selected in SimVision. Similarly, if you highlight a primitive in the hierarchy editor, the corresponding primitive in the schematic editor appears highlighted as well, and vice versa.

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.



```
(amslib top schematic)
  I0 (amslib signalSrc module)
  I1 (amslib samplehold module)
  I2 (amslib comparator schematic)      sche
  I3 (diglib sareg module)
  I4 (amslib daconv module)
  I5 (diglib vhdl_clock vhdl_behaviora
  V0 (analogLib vdc spectre)
```

```
sareg  (* integer library_binding = "digli
  .endOfConv( endOfConv ), .b5( b5 ), .b6( 
  .b0( b0 ), .clkSig( clkSig ), .b4( b4 ),
  .result( compOut ), .trigger( endOfConv )

daconv #( refVolt(5.000000))  (* integer l
  *) I4 ( .b2( b2 ), .b5( b5 ), .b6( b6 ),
  .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” on page 485. With these options set, you can cross-probe nets as summarized in the following table.

		Responding Application		
		Schematic Editor	SimVision	Waveform
Initiating Application	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.

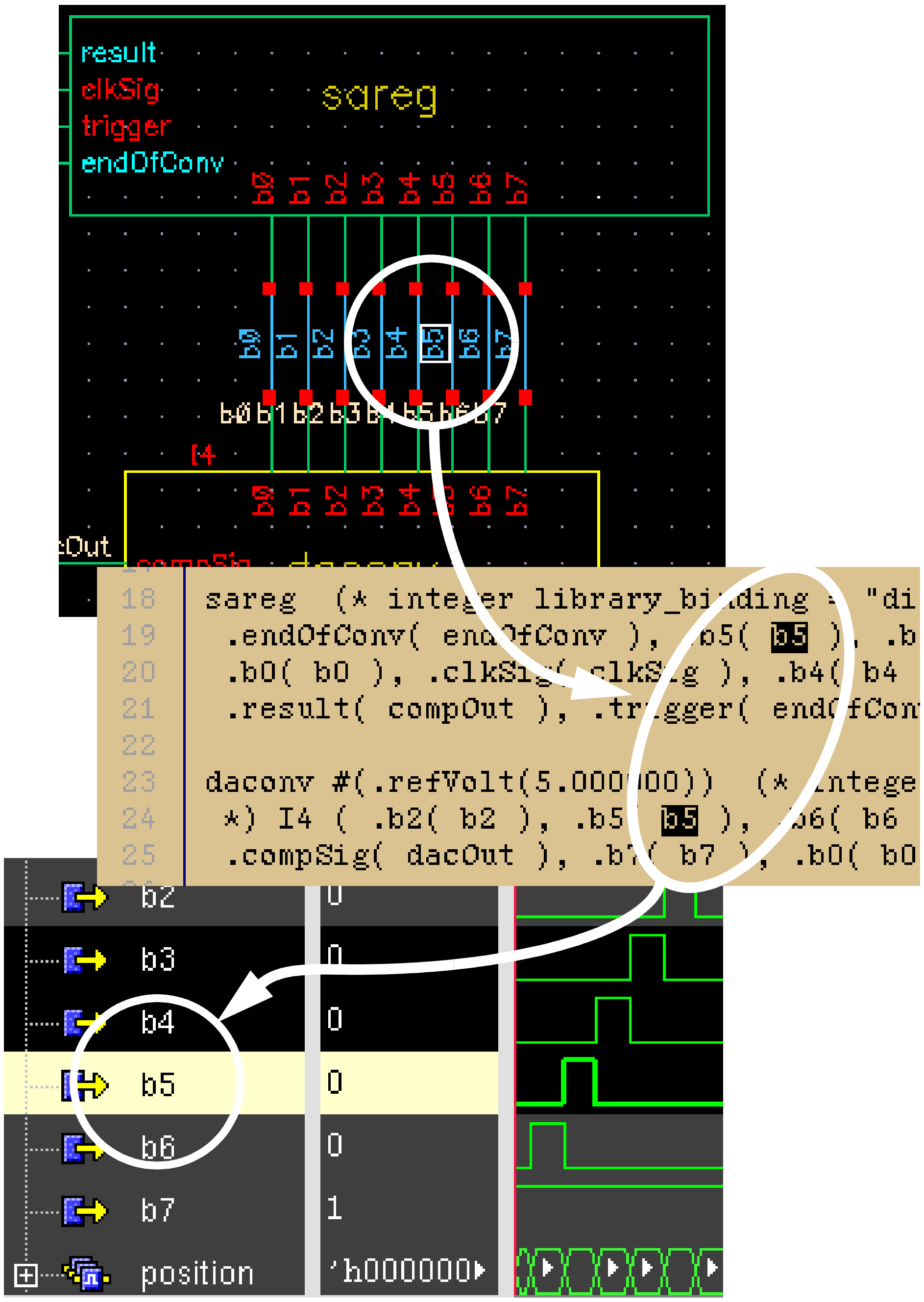
You cannot cross-probe global signals with AMS Designer.

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.



```
sareg  (* integer library_binding = "di
 .endOfConv( endOfConv ), .b5( b5 ), .b
 .b0( b0 ), .clkSig( clkSig ), .b4( b4
 .result( compOut ), .trigger( endOfCon

daconv #(.refVolt(5.000000))  (* intege
 *) I4 ( .b2( b2 ), .b5( b5 ), .b6( b6
 .compSig( dacOut ), .b7( b7 ), .b0( b0
```

```
b2        0
b3        0
b4        0
b5        0
b6        0
b7        1
position  'h000000
```

17 Debugging

This chapter contains the following topics:

- Managing Custom Buttons on page 521

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. See also Chapter 16, "Using SimVision with the AMS Simulator."
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 `analog` block.
digital context	The context of statements that appear in a location other than an `analog` block.

Managing Databases

You can open, close, disable, enable, and display information about databases. See the following topics for more information:

- Opening a Database on page 493
- Displaying Information about Databases on page 493
- Disabling a Database on page 494
- Enabling a Database on page 494
- Closing a Database on page 495

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.

Note: 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.

Note: 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.

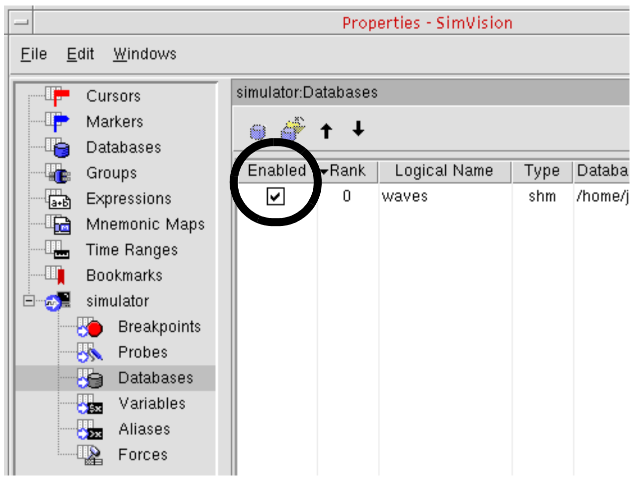
  Syntax:

  ```
  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.



Enabled	Rank	Logical Name	Type	Databa
✓	0	waves	shm	/home/j

  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.

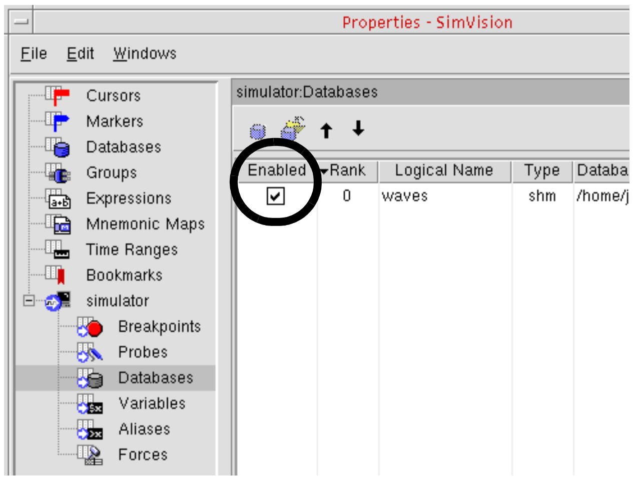
  Syntax:

  ```
  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.



Enabled	Rank	Logical Name	Type	Databa
✓	0	waves	shm	/home/j

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*.

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.

  See "Creating and Managing Probes" in the *Running SimVision in Simulation Mode* book for examples.

Note: For Verilog-AMS, digital objects can be probed only if they have read access. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to digital simulation objects.

See the following topics for Tcl command syntax:

- Setting a Probe Using the Tcl probe Command on page 496
- Displaying Information about Probes Using the Tcl probe Command on page 497
- Disabling a Probe Using the Tcl probe Command on page 497
- Enabling a Probe Using the Tcl probe Command on page 497
- Deleting a Probe Using the Tcl probe Command on page 497

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]
         [-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} ...
```

Traversing the Model Hierarchy

The Virtuoso 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`)

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:

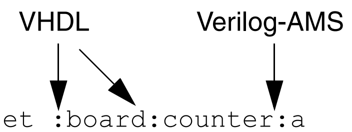
```
:vending
:vending:drinks
:vending:drinks:sig2
```

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 Virtuoso 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:



```
ncsim> scope -set :board:counter:a
ncsim> scope -set board:counter.a
ncsim> scope -set board.counter:a
ncsim> 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.

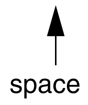
Record field specifiers apply only to VHDL objects. Use a period to separate the object name from the record field.

The following pairs of Tcl commands are identical.

```
ncsim> scope foo_array(2)
ncsim> scope foo_array[2]

ncsim> value sig[7:0]
ncsim> value sig(7:0)

ncsim> value sig[7]
ncsim> value sig(7)

ncsim> describe sig[7 downto 0]
ncsim> 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.\szome_name .signal
```



For VHDL, escaped names begin and end with a backslash (for example, `\w3.OUT\`).

The following two `value` commands are identical:

```
ncsim> value top.vending.@{\w3.OUT }

ncsim> value top.vending.@{\w3.OUT\}
```

Setting 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. See “Tcl Expressions as Arguments” on page 611 for details on the syntax of these expressions.

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. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.

See "Disabling, Enabling, Deleting, and Displaying Breakpoints" on page 505 for more information on breakpoints.

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. (See "-LINedebug" in the "Compiling Verilog Source Files with ncvlog" chapter of the *Compiling Verilog Source Files* book for details on using this 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*.

See "Disabling, Enabling, Deleting, and Displaying Breakpoints" on page 505 for more information on breakpoints.

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. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 443 for details on specifying access to simulation objects.

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:

```
ncsim> 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*.

See “Disabling, Enabling, Deleting, and Displaying Breakpoints” on page 505 for more information on breakpoints.

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.

Note: 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*.

See "Disabling, Enabling, Deleting, and Displaying Breakpoints" on page 505 for more information on breakpoints.

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*.

See "Disabling, Enabling, Deleting, and Displaying Breakpoints" on page 505 for more information on breakpoints.

Disabling, Enabling, Deleting, and Displaying 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 to 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.

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.

Note: 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.

      Note: 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*.

  See Introduction to SimVision for other Simulation menu choices and information about simulation toolbar buttons.

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 `ncelab`. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 443 for details on specifying access to simulation objects.

To force 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.

  Note: 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.

  Note: 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.

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 `ncelab`. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 443 for details on specifying access to simulation objects.

Note: 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.

Note: 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:

  ```
  ncsim> 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.

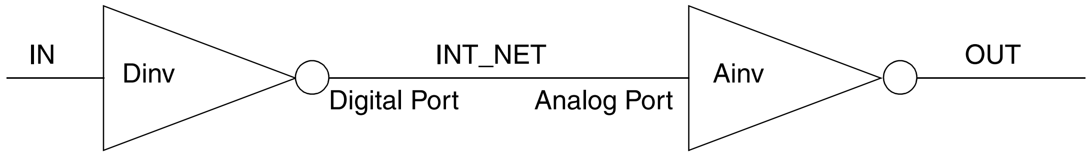
- If you are using SimVision, see "Tracing Paths with the Trace Signals Sidebar" in the *Tracing Signal Values* book.

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. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.

Debugging Designs with 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.

Displaying Waveforms in the Waveform Window

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.

Note: 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. See

- "Managing Databases" on page 492 for details on opening a database.
  This topic also contains information on displaying information about databases and on disabling, enabling, and closing a database.
- "Setting and Deleting Probes" on page 496 for details on probing signals.

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
`$shm_open();`	Opens a simulation database.
`$shm_probe();`	Selects signals whose simulation value changes will enter the simulation database.
`$shm_close;`	Closes a simulation database.

Example:

```
initial
    begin
         $shm_open("waves.shm");
         $shm_probe();
         #1 $stop; // stop simulation at time 1
    end
```

Tip

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] ) ;
```

"*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

Note: 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.

See the following books for details:

- Using the Design Browser
- Running SimVision in Simulation Mode

If you do not preselect objects, you can use the Design Browser to probe signals and add them to the Waveform window.

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:

  ```
  ncsim> database -show
  ```

  ```
  ncsim> probe -show
  ```

  Use the `alias` command without a modifier to display information about aliases you have set, as shown in the following example:

  ```
  ncsim> alias
  e      exit
  f2     finish 2
  h      history
  ncsim>
  ```

- 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:

  ```
  ncsim> set abc 10
  ncsim> set vlog_format %b
  ```

- Delete a variable using the `unset` command:

  ```
  ncsim> unset abc
  ```

- Display a list of predefined simulation variables and their current values using the `help -variables` command:

  ```
  ncsim> help -variables
  ```

- Display a list of all currently set variables using the `info vars` command:

  ```
  ncsim> info vars
  ```

  Note: This command does not display variable values.

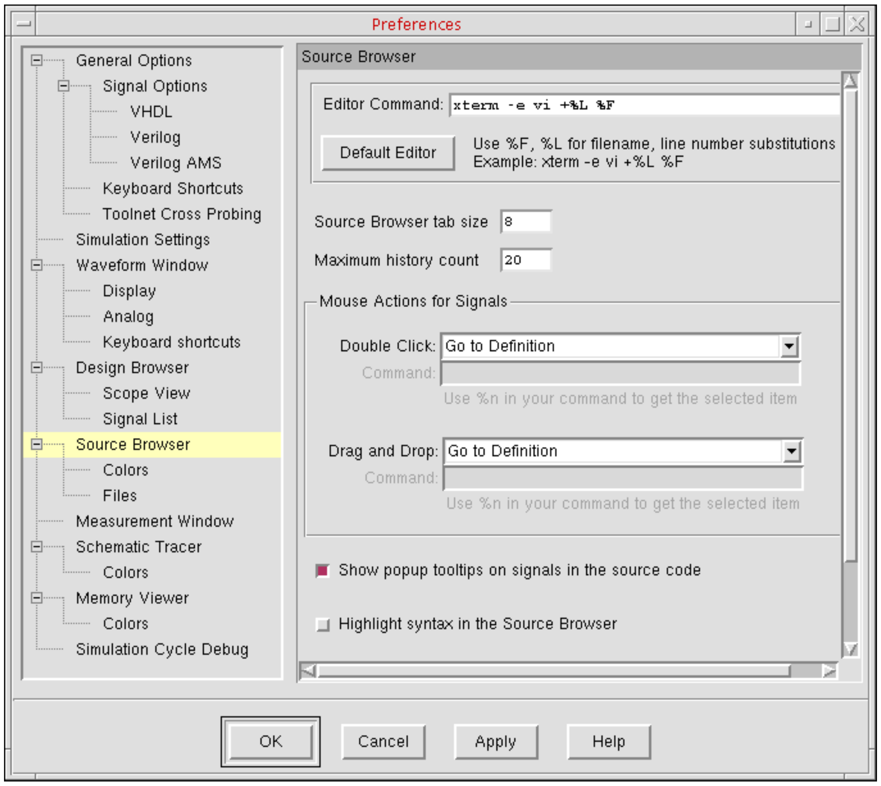
See also “Controlling the Simulation” in the *Running SimVision in Simulation Mode* book.

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.



Editor Command: `xterm -e vi +%L %F`

Use %F, %L for filename, line number substitutions
Example: xterm -e vi +%L %F

Source Browser tab size: 8

Maximum history count: 20

Double Click: Go to Definition
Command:
Use %n in your command to get the selected item

Drag and Drop: Go to Definition
Command:
Use %n in your command to get the selected item

[✓] Show popup tooltips on signals in the source code

[ ] Highlight syntax in the Source Browser

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.

See also "Accessing Design Source Code" in the *Using the Source Browser* book.

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.

For more information, see “Accessing Design Source Code” in the *Using the Source Browser* book.

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:

    ```
    ncsim> 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.

  For more information, see "Saving and Restoring Your Debugging Environment" in the *Running the SimVision Analysis Environment* book.

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

Important

You cannot create an alias with the same name as a predefined Tcl command.

To create and 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

See "Customizing Toolbars" in the *Introduction to SimVision* book for details and examples.

A

Updating Legacy Libraries and Netlists

This appendix highlights changes that you might have to make to your existing libraries before using them with the AMS Designer simulator.

Updating 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 the "Updating Verilog-A Modules" appendix, in *Cadence Verilog-AMS Language Reference*.

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 the "Mixed-Signal Aspects of Verilog-AMS" chapter, in *Cadence Verilog-AMS Language Reference*.

Updating SpectreHDL Modules

The AMS Designer simulator does not support SpectreHDL modules, so to use that functionality, you must convert SpectreHDL modules to Verilog-A. In most cases, conversion involves syntax changes only, but sometimes semantic and functional differences prevent conversion. See "Converting SpectreHDL to Verilog-A" in the *Cadence Verilog-A Language Reference* for more information.

Updating 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 Chapter 6, "Preparing the Design: Using Analog Primitives and Subcircuits," for more information.

Updating Verilog Modules

A module written for the purely digital Verilog language can often be used without change 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 `ncvlog` 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.

Updating VHDL Blocks

With the Virtuoso 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.

Updating 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 Virtuoso® 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 Chapter 4, "Specifying Controls for the Analog Solvers."

The AMS Designer simulator does not support everything used in existing netlists. For information about unsupported features, see the *Virtuoso AMS Simulator Known Problems and Solutions*.

Updating 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 the above mentioned issues 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*.

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

```
ncsim> Tcl_command [-modifiers] [-options] [arguments]
```

Note: See also important information in "Specifying Unnamed Branch Objects" on page 620.

For information on the Tcl commands you can use with the Virtuoso® AMS Designer simulator, see the following table. Some commands are also links to more information. For commands that are not links, and for commands you can use only with purely digital designs, you can find more information in "Using the Tcl Command-Line Interface" in the *Incisive Simulator Tcl Command Reference*.

Tcl Command	**Description**
`alias`	Defines aliases that you can use as command short-cuts.
`analog`	Controls the analog solver during mixed-signal simulation using AMS simulator with Spectre/APS solver.
`attribute`	Enables VHDL function-valued attributes for specified signals so that they can be accessed from the Tcl interface with the `value` command. The `attribute` command is enabled only for purely digital designs.
`call`	Lets you call a user-defined C-interface function or a Verilog user-defined PLI system task or function from the command line.
`coverage`	For purely digital designs, controls the dumping of code coverage data.
`database`	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.

Tcl Command	Description
`deposit`	Lets you set the value of an object.
`describe`	Displays information about the specified simulation object, including its declaration.
`drivers`	Displays a list of all contributors to the value of the specified objects.
`exit`	Terminates simulation and returns control to the operating system.
`finish`	Causes the simulator to exit and returns control to the operating system.
`fmibkpt`	Performs operations on breakpoints that are coded into C models using the `fmiBreakpoint` call.
`force`	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.
`help`	Displays information about simulator (`ncsim`) commands and options and predefined variable names and values.
`history`	Lets you reexecute commands without having to retype them.
`input`	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.
`memory`	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.
`omi`	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.
`probe`	Lets you control the values being saved to a database.
`process`	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.
`release`	Releases any force set on the specified objects.
`reset`	Resets the currently loaded model to its original state at time zero. This command is enabled only for purely digital designs.

Tcl Command	Description
`restart`	Replaces the currently simulating snapshot with another snapshot of the same elaborated design. You cannot use `restart` after the current AMS simulation is started.
`run`	Starts simulation or resumes a previously halted simulation.
`save`	Creates a snapshot of the current simulation state.
`scope`	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.
`source`	Lets you execute a file containing simulator commands.
`stack`	Lets you view or set the current stack frame. This command is enabled only for purely digital designs.
`status`	Displays memory and CPU usage statistics and shows the current simulation time.
`stop`	Creates or operates on a breakpoint.
`strobe`	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.
`task`	Lets you schedule Verilog-AMS tasks for execution. This command is disabled while the analog solver is active.
`time`	Displays the current simulation time scaled to the specified unit.
`value`	Prints the current value of the specified objects using the last format specifier preceding the object name argument.
`version`	Displays the version number of `ncsim`.
`where`	Displays the current location of the simulation.

analog

Controls the analog solver in a mixed-signal simulation.

Syntax

```
Tcl> analog [-stop time] [-show] [ -reltol r_val ] [ -iabstol i_val ]
        [ -vabstol v_val ]
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-stop` *`time`*	Changes the analog stop time to *`time`*. If *`time`* is not 1% larger than the current analog stop time, or if *`time`* is larger than 1000 times the original stop time in the analog control file, the simulator ignores the time with a warning.
	`-show`	Prints the present values of the analog solver options and indicates which options can be set using the `analog` command. This option also shows the values, if any, that will be set for the next analog time step.
	`-reltol` *`r_val`*	Changes the reltol of the analog solver. *`r_val`* must be less than 0.1 when the `errpreset` option is set to `liberal`, but must always be less than 1.
	`-iabstol` *`i_val`*	Changes the current abstol of the analog solver.
	`-vabstol` *`v_val`*	Changes the voltage abstol of the analog solver

See also "How to Use save, restart, and analog" on page 582.

call

Calls a user-defined or predefined VPI or system task or function from the command line.

Note: In this release, you cannot call an analog VPI or system task or function. In addition, you cannot use the `call` command for digital tasks or functions while the analog solver is active.

Syntax

`call` `[`**`-systf`** `|` **`-predefined`**`]` *`task_or_function_name`* `[`*`arg1`* `[`*`arg2`* `...]]`

If you use the `-systf` or `-predefined` command option, the option must appear before the task or function name or the simulator interprets it as an argument to the task or function. See "Modifiers and Options" on page 533 for details on these options.

The *`task_or_function_name`* argument is the name of the system task or function with or without the beginning dollar sign. The dollar sign character has a special meaning in Tcl. If the name of the task or function contains any dollar signs, you must enclose the argument in curly braces or precede each dollar sign by a backslash. For example, you can start a system task or function called `$mytask` with

```
ncsim> call mytask
ncsim> call \$mytask
ncsim> call {$mytask}
ncsim> call {mytask}
```

You can start a system task or function called `$my$task` with any of the following:

```
ncsim> call my\$task
ncsim> call \$my\$task
ncsim> call {$my$task}
ncsim> call {my$task}
```

Arguments to the system task or function can be either literals or names.

Literals can be:

- Integers

  ```
  ncsim> call mytask 5
  ncsim> call mytask 5 7
  ```

- Reals

  ```
  ncsim> call mytask 3.4
  ncsim> call mytask 22.928E+10
  ```

- Strings

  Strings must be enclosed in quotation marks. Enclose strings in curly braces or use the backslash character to escape quotation marks, spaces, and other characters that have special meaning to Tcl. For example:

  ```
  ncsim> call mytask {"hello world"}
  ncsim> call mytask \"hello\ world\"
  ```

- Verilog literals, such as `8’h1f`

Names can be full or relative path names of instances or objects. Relative path names are relative to the current debug scope (set by the `scope` command). Object names can include a bit select or part select. For example:

```
ncsim> call  mytask top.u1
ncsim> 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:

```
ncsim> call \$mytask a+b
ncsim> call \$mytask {func a}
```

However, literals can be created using the Tcl `expr` command. For example, if the desired argument is the expression `(a+b)`, use the following:

```
ncsim> 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.

Note: 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:

```
ncsim> call task [call func arg1 ...]
ncsim> force a = [call func arg1 ...]
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-systf`	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 `call` command is also used to start 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 `-systf` 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 `-predefined` option. The command `call -systf` with no task or function name argument displays a list of all registered user-defined system tasks and functions.
	`-predefined`	Looks for the specified task or function name only in the table of predefined CFC library functions. You cannot use the `-predefined` 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 `-systf` option. The command `call -predefined` with no function name argument displays a list of all predefined C function names.

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:

```
ncsim> call \$hello_task
```

This task can also be started with any of the following:

```
ncsim> call hello_task
ncsim> call {$hello_task}
ncsim> call {hello_task}
```

The `$hello_func` function can be started with any of the following commands:

```
ncsim> call \$hello_func
ncsim> call hello_func
ncsim> call {$hello_func}
ncsim> 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.

```
ncsim> call hello_task [call hello_func]
```

The following command displays a list of all registered user-defined system tasks and functions.

```
ncsim> call -systf
```

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.

Note: 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 transactions 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 "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.

The `deposit` command is supported on user-defined net types. For more information on user-defined net types, refer to User-Defined Net Type and Resolution Function on page 262

Syntax

```
deposit object_name [=] value
    [-after time_spec {-relative | -absolute}]
    [-inertial]
    [-transport]
    [-generic]
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-after` *`time_spec`*	Causes the assignment to occur at a time in the future, rather than immediately. *`time_spec`* can be relative (the default) or absolute. 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.
	`-absolute`	Causes the assignment to occur at the simulation time specified in *`time_spec`*.
	`-relative`	Causes the assignment to occur after the amount of time specified in *`time_spec`* has passed. This is the default.
	`-inertial`	Deposits the value after an inertial delay.
	`-transport`	Deposits the value after a transport delay.
	`-generic`	Deposits generic value. This operation might lead to violation of globally static bounds.

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.

```
ncsim> deposit r[0:7] = 8’h1F
```

The following command assigns `25` to `r[8:15]` after simulation resumes and 1 time unit has elapsed.

```
ncsim> deposit r[8:15] = 25 -after 1
```

The following command assigns `25` to `r[8:15]` at simulation time 1 ns.

```
ncsim> 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.

```
ncsim> 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`.

```
ncsim> 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.

```
ncsim> deposit clrb 1
ncsim: *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.

```
ncsim> deposit :t_nickel_out = ’1’
```

The following command deposits the value 1 to object `:top:DISPENSE_tempsig` (std_logic).

```
ncsim> deposit :top:DISPENSE_tempsig ’1’
```

The following command deposits the value 0 to object `:t_dimes` (std_logic_vector) after 10 ns has elapsed.

```
ncsim> deposit -after 10 ns -relative :t_DIMES {"00000000"}
```

The following command deposits the value TRUE to object `stoppit` (boolean).

```
ncsim> deposit stoppit true
```

The following command deposits the value 10 to object `:count` (integer).

```
ncsim> deposit :count 10
```

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.

For information about using `describe` with unnamed branches, see "Specifying Unnamed Branch Objects" on page 620.

See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.

The `describe` command is supported on user-defined net types. For more information on user-defined net types, refer to User-Defined Net Type and Resolution Function on page 262.

Syntax

```
describe simulation_object ...
```

Modifiers and Options

None.

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.

```
ncsim> 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.

```
ncsim> 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,

```
ncsim> 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`.

```
ncsim> describe data
data......register [3:0] = 4'h0
```

The following command displays information about two Verilog objects: `data` and `q`.

```
ncsim> 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`.

```
ncsim> describe alu_16
alu_16.....top-level module
```

The following command displays information about the object `u1`.

```
ncsim> describe u1
u1.......instance of module arith
```

The following commands display information about the mixed bus `w`.

```
ncsim> describe w
w.........wire (mixed bus) = Inf
ncsim> describe w[0]
w.........wire (electrical)
ncsim> describe w[1]
w.........wire [0:2]
w[1] (wire/tri) = StX
ncsim> describe w[2]
w.........wire [0:2]
w[2] (wire/tri) = StX
```

The following command displays information about the connect module `mya2d`.

```
ncsim> describe mya2d
mya2d........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)`.

```
ncsim> describe d
d.........input [3:0]
```

```
    d[3]  (-RW)
    d[2]  (-RW)
    d[1]  (-RW)
    d[0]  (-RW)
```

VHDL examples:

```
ncsim> describe t_NICKEL_IN
t_NICKEL_IN...signal : std_logic = ’0’
ncsim> describe t_NICKEL_IN t_CANS
t_NICKEL_IN...signal : std_logic = ’0’
t_CANS........signal : std_logic_vector(7 downto 0) = "11111111"
ncsim> describe gen_dimes
gen_dimes...process statement
ncsim> describe :top
top........component instantiation
```

drivers

Displays a list of all contributors to the value of the specified digital objects.

Note: 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" on page 585 for details on the `scope` command.

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. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.

See "drivers Command Report Format" on page 543 for details on the output format of the `drivers` command. See "Examples" on page 546 for examples.

The `drivers` command is supported on user-defined net types. For more information on user-defined net types, refer to User-Defined Net Type and Resolution Function on page 262

Syntax

```
drivers object_name ...
    [-effective]
    [-future]
    [-novalue]
    [-verbose]
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-effective`	Displays contributions to the effective value of the signal. By default, the `drivers` command displays contributions to the driving value. Only VHDL inout and linkage ports can have different driving and effective values.

Modifiers	Options and Arguments	Function
	`-future`	Displays the transactions that are scheduled on each driver.
	`-novalue`	Suppresses the display of the current value of each driver.
	`-verbose`	**Note:** This option affects VHDL signals only. 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 `-verbose` option, resolution and type conversion function information is omitted from the output.

drivers Command Report Format

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 *source_description*. This implies that a sequential signal assignment statement within a process is the driver. The *scope_name* is the scope name of the process.

Concurrent Signal Assignment/Concurrent Procedure call

The *source_description* 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 *scope_name.*

No drivers

If the signal has no drivers, the text `No drivers` appears verbatim.

A Verilog driver

If the driver is from a Verilog model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a Verilog model.
```

This report indicates that the signal is ultimately driven by a port (connected to *port_name* 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:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a C model.
```

Driver from a LMC model

If the driver is from an imported LMC model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_file_containing_entity], driven by a LMC model.
```

Driver from an OMI model

If the driver is from an imported OMI model, the report has the following form:

```
port 'port_name' in entity(arch) [File:
    path_to_shell_file], driven by a OMI model.
```

Resolution / Type Conversion Function in Non-Verbose mode

If you do not use the `-verbose` option, the text `[verbose report available ....]` may appear. This indicates that the signal gets its value from a resolution function or a type conversion function. Use `-verbose` 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 `-verbose` option is used:

```
[resolution function function_name()]
```

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 `-verbose` option is used:

```
[type conversion function function_name(formal)]
```

This means that the signal's driving value comes from a type conversion function on a formal 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 `-verbose` option is used:

```
[type conversion function function_name(actual)]
```

This means that the signal's effective value comes from a type conversion function on an actual 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 `GUARD` expression:

```
[implicit guard signal]
```

Signal Attribute

The following is displayed in response to a query on an IN port that ultimately is associated with a signal valued attribute:

```
[attribute of signal full_path_of_the_signal]
```

`full_path_of_the_signal` corresponds to the complete hierarchical path name 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:

```
[constant expression associated with port port_name]
```

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.

Examples

This section includes examples of using the `drivers` command with digital Verilog-AMS and with VHDL signals.

Example Output for Digital Verilog-AMS Signals

The following command lists the drivers of a signal called `f`.

```
ncsim> 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`.

```
ncsim> 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`.

```
ncsim> 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`.

```
ncsim> 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`.

```
ncsim> drivers w
ncsim: *E,MIXBOFF: Mixed discipline bus ’w’ needs an index.
ncsim> drivers w[0]
ncsim: *E,TNODIA: No drivers exist for analog object: top.w[0].
ncsim> drivers w[1]
w.........wire [0:2]
w[1] (wire/tri) = StX
No drivers
ncsim> 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 `ncsim` 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.

```
ncsim> drivers count
ncsim: *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:

```
ncsim> drivers :u1.a

u1.a.......input (wire/tri) = St1
    St1 <- (:u1) driven by a VHDL model

ncsim> 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

ncsim>
```

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:

```
ncsim> 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

Example 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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`.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> drivers -effective i1:a
i1:a.......port : in std_logic = ’1’
    ’1’ <- (and2_top.i1) driven by a Verilog model

ncsim> drivers -effective i1:i1:port1
i1:i1:port1...port : in std_logic = ’1’
    ’1’ <- (and2_top.i1) port ’a’ in and2(and2_bot) [File:
         ./and2.vhd], driven by a Verilog model
```

finish

Closes the simulator and returns control to the operating system.

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.

See "Exiting the Simulation" on page 470 for more information.

Syntax

```
finish [0 | 1 | 2]
```

Modifiers and Options

None.

Examples

The following command ends the simulation session and prints the simulation time.

```
ncsim> 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.

```
ncsim> 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
%
```

force

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. (See “release” on page 566 for details on the `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.

Note: 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 that is being forced must have write access. An error is printed if it does not. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 443 for details on specifying access to simulation objects.

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.

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.
- Bit-select or part-select of a wreal array cannot be forced.
- A wreal array cannot be forced.
- Out of module references (OOMRs) to wreals that require support of Hierarchical IE optimization and IE insertion in VHDL scope are not supported.

Forces created by the `force` command and those created by a Verilog `force` procedural statements are saved if the simulation is saved.

See the "Forcing and Releasing Signal Values" section in the "Debugging Your Design" chapter of *Cadence Verilog Simulation User Guide* for more information.

See the "Basics of Tcl" appendix in *Cadence Verilog Simulation User Guide* for information on using Tcl with NC-Verilog.

Syntax

```
force object_name [=] value
```

Modifiers and Options

None.

Examples

Digital Verilog-AMS examples:

The following command forces object `r` to the value `'bx`. The equal sign is optional.

```
ncsim> force r = 'bx
```

The following command uses value substitution. Object `x` is forced to the current value of w.

```
ncsim> 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`.

```
ncsim> 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.

```
ncsim> force clrb 1
ncsim: *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.

```
ncsim> force :t_nickel_out = '1'
```

The following command forces object `:top:DISPENSE_tempsig` (std_logic) `to 1.`

```
ncsim> force :top:DISPENSE_tempsig '1'
```

The following command forces object `:t_dimes` (std_logic_vector) to 0.

```
ncsim> force :t_DIMES {"00000000"}
```

The following command forces object `is_ok` (boolean) to TRUE.

```
ncsim> force :is_ok true
```

The following command forces object `:count` (integer) to 10.

```
ncsim> force :count 10
```

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`)

Note: 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" on page 531 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. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.

Syntax

Note: 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 *Incisive Simulator Tcl Command Reference*.

You can use the `probe` command with the following modifiers, options, and arguments with the AMS Designer simulator:

```
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]
    |-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
    |-inhconn_signal global_signal
    |-name probe_name
    |-screen [-format format_string] [-redirect filename] objects
    |-variables
    |-waveform

db_format::=
     -shm
    |-vcd
    |-evcd
    |-database dbase_name
```

The argument to `-delete`, `-disable`, `-enable`, or `-show` can be:

- A probe name
- A list of probe 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 probe names and patterns

Modifiers

The `probe` command has the following modifiers:

Modifier	Cross-Reference
`[-create][ {`*object*`|`*scope_name*`} ... ][ `*options*`]`	See “-create” on page 558
`-delete {`*probe_name*`|`*pattern*`} ...`	See “Deleting a Probe” in the “probe” section of “Using the Tcl Command-Line Interface” in the *Incisive Simulator Tcl Command Reference*.
`-disable {`*probe_name*`|`*pattern*`} ...`	See “Disabling a Probe” in the “probe” section of “Using the Tcl Command-Line Interface” in the *Incisive Simulator Tcl Command Reference*.
`-enable {`*probe_name*`|`*pattern*`} ...`	See “Enabling a Probe” in the “probe” section of “Using the Tcl Command-Line Interface” in the *Incisive Simulator Tcl Command Reference*.
`-save [`*filename*`]`	See “Saving a Script to Re-Create Probes” in the “probe” section of “Using the Tcl Command-Line Interface” in the *Incisive Simulator Tcl Command Reference*.
`-show [{`*probe_name*`|`*pattern*`} ...][-database `*dbase_name*`]`	See “Displaying Information about Probes” in the “probe” section of “Using the Tcl Command-Line Interface” in the *Incisive Simulator Tcl Command Reference*.

-create

The optional `-create` modifier for the `probe` command has the following general syntax:

`[-create][ {`*`object`*`|`*`scope_name`*`} ... ][ `*`options`*`]`

Use the `-create` modifier to place values of the specified simulation objects in a database.

Note: 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, refer to the section Probe Options Supported in SPICE Scope on page 565.

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 AMS Designer simulator:

- `-aicms`
- `-all [-variables]`
- `-database` *`dbase_name`*

- `-depth` {*n* | `all` | `to_cells`}

  Note: The software does not probe currents through inherited connections when processing the `all` argument.

- `-domain` {`analog` | `digital`}
- `-emptyok`
- `-evcd`
- `-flow`
- `-inhconn_signal` *`global_signal`*
- `-inputs`
- `-name` *`probe_name`*
- `-outputs`
- `-ports`
- `-screen [-format` *`format_string`*`] [-redirect` *`filename`*`]` *`objects`*
- `-shm`
- `-vcd`

For information about these and other options for the `-create` modifier, see "Creating a Probe" in the "probe" section of "Using the Tcl Command-Line Interface" in the *Incisive Simulator Tcl Command Reference*.

You can use the `probe -create -screen` and `probe -create -shm` commands to probe objects with built-in and user-defined net types 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
- Net types of type unpacked structure with only real as sub elements
- Unpacked array of net types with elements as scalar real with built-in and user-defined resolution functions
- Unpacked array of unpacked structure net type with built-in and user-defined resolution functions.
  - Probe on the complete bit select on the array element is supported
  - Probe on the member select of the bit select of the array element is supported

For more information on user-defined net types, refer to User-Defined Net Type and Resolution Function on page 262

-aicms

Use the `-aicms` option to probe 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`. (See “Syntax” on page 556 for more information about `probe` command syntax.)

You cannot use the following options when you specify `-aicms`:

- `-inhconn_signal` *`global_signal`*
- `-evcd`
- `-screen [-format` *`format_string`*`] [-redirect` *`filename`*`]` *`objects`*
- `-vcd`

For more information about these and other options for the `-create` modifier, see “Creating a Probe” in the “probe” section of “Using the Tcl Command-Line Interface” in the *Incisive Simulator Tcl Command Reference*.

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
```

-flow

The `-flow` option 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.

Note: 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

Specify 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.

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 `ncsim.shm` in the file `ncsim.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.

```
ncsim> probe -create -shm -all
```

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 `ncsim.vcd` in the file `ncsim.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.

```
ncsim> 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.

```
ncsim> 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 `ncsim.shm` in the file `ncsim.shm`, if necessary).

```
ncsim> 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 `ncsim.shm` in the file `ncsim.shm`, if necessary).

```
ncsim> 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 `ncsim.shm` in the file `ncsim.shm`, if necessary).

```
ncsim> probe -shm u1.sum
```

The following command creates a probe on all objects in scope `u1`.

```
ncsim> probe -shm u1
```

The following command creates a probe on all objects in scopes `u1` and `u2`.

```
ncsim> probe -shm u1 u2
```

The following command creates a probe on all ports in scope `u1`.

```
ncsim> probe -shm u1 -ports
```

The following command creates a probe on all ports in scope `u1` and its subscopes.

```
ncsim> probe -shm u1 -ports -depth 2
```

The following command creates a probe on all ports in scope `u1` and all scopes below `u1`.

```
ncsim> probe -shm u1 -ports -depth all
```

The following command creates a probe on every signal inside spice instance `spice_A1`.

```
ncsim> 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.

```
ncsim> probe -shm u1 -ports -depth to_cells
```

The following command creates a probe called `peek`.

```
ncsim> 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.

```
ncsim> probe -screen clock count analogsig
Created probe 1
ncsim> 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`.

```
ncsim> probe -screen -format "clock = %d \ncount = %b" clock count
Created probe 1
ncsim> run 10 ns
clock = 1'd1
count = 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.

```
ncsim> probe -screen clock count
Created probe 1
ncsim> probe -disable 1
ncsim> 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.

```
ncsim> probe -show
```

The following command displays the state of the probe called `peek`.

```
ncsim> probe -show peek
```

The following command disables the probe called `peek`.

```
ncsim> probe -disable peek
```

The following command enables the probe called `peek`, which was disabled in the previous command.

```
ncsim> probe -enable peek
```

The following command deletes all probes beginning with the characters `pe`.

```
ncsim> probe -delete pe*
```

The following command deletes all probes beginning with the characters `v` and `w`.

```
ncsim> 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.

```
ncsim> probe -shm d
ncsim: *E,RDACRQ: Object does not have read access: hardrive.h1.d.
```

The following command produces a warning message for attempting to probe an analog object to a VCD database.

```
ncsim> probe -create -vcd top.analogResult -waveform
```

The error message is

```
ncsim: *W,PRALOB: Cannot probe analog object:
         top.analogResult. This object ignored.
```

```
ncsim: *E,PRWHAT: no items specified in probe -create command.
```

Probe Options Supported in SPICE Scope

The following table displays the probe options that can be used to probe signals inside the SPICE hierarchy.

Table B-1 Probe Options Supported in SPICE Scope

`-create`	`-shm`	`-database` *`dbase_name`*	`-all`
`-depth`	`-domain`	`-emptyok`	`-exclude`
`-flow`	`-name` *`probe_name`*	`-ports`	`-screen [-format] [-redirect]`
`-waveform`	`-delete`	`-disable`	`-enable`
`-save`	`-show`		

Note: Tcl probing inside the SPICE hierarchy is not supported in AMS-UltraSim. In addition, Tcl probing inside the SPICE hierarchy is not supported on the AIX platform.

release

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.

Note: 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. See “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 443 for details on specifying access to simulation objects.

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

See the “Forcing and Releasing Signal Values” section in the “Debugging Your Design” chapter of *Cadence Verilog Simulation User Guide* for more information

Syntax

```
release object_name ...
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-keepvalue`	Releases the forced object, but retains the forced value.

Examples

The following command removes a force set on object `x`.

```
ncsim> release x
```

The following command removes a force set on object `:top:DISPENSE_tempsig`.

```
ncsim> release :top:DISPENSE_tempsig
```

The following command releases two objects: `w[0]` and `r`.

```
ncsim> 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.

```
ncsim> release b6
*E,SETAIA: Analog engine is active. Cannot release digital object: top.sar.b6
```

reset

Resets the currently loaded model to its original state at time zero. The time-zero snapshot, created by the elaborator, must still be available.

Note: 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.

  Note: 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.

Syntax

```
reset
```

Modifiers and Options

None.

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.

```
ncsim> 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.

```
ncsim> reset
*E,RESTAG: Reset not supported for mixed-signal designs.
```

restart

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 `ncsim` 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:

`restart top`	Restarts [*lib*.]top[:*view*] 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.
`restart top:ckpt`	Restarts [*lib*.]`top:ckpt`
`restart :ckpt`	Restarts [*lib*.][*cell*]`:ckpt`

The following restrictions apply to 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 `ncsim` 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.

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.

  Note: 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,

```
ncsim top
      (open a database, set probes, set breakpoints, deposits, forces, etc.)
ncsim> run 100 ns
ncsim> save worklib.top:ckpt1
ncsim> save -environment ckpt1.tcl
ncsim> exit
ncsim -tcl worklib.top:ckpt1
ncsim> source ckpt1.tcl
```

Syntax

```
restart snapshot_name
```

```
restart -show
```

Modifiers and Options

Modifiers	Options and Arguments	Function
`-show`		Lists the names of all snapshots that can currently be used as the argument to the `restart` command.

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.

```
ncsim> save top:ckpt1
ncsim> restart top:ckpt1
ncsim> 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.

```
ncsim> save top:ckpt1
ncsim> save -environment top_ckpt1.env
ncsim> exit
ncsim -tcl :ckpt1
ncsim> 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.

```
ncsim> restart top
```

The following command lists all of the snapshots you can currently load with the `restart` command.

```
ncsim> restart -show
otherlib.board:module
worklib.board:ckpt1
worklib.board:ckpt2
```

For examples illustrating how the `save`, `restart`, and `analog` commands work together, see "How to Use save, restart, and analog" on page 582.

run

Starts simulation or resumes a previously halted simulation.

Using the `run` command, you can perform the following tasks:

Task	**run Option**
Run until it is possible to create a snapshot with the `save -simulation` command.	`-clean`
Run until an interrupt, such as a breakpoint or error, occurs or until simulation completes.	(None)
Run one behavioral statement, stepping over subprogram calls.	`-next`
Run one behavioral statement, stepping into subprogram calls.	`-step`
Run until the current subprogram (task, function, procedure) returns.	`-return`
Run to a specified timepoint or for a specified number of time units.	`-timepoint`
Run to the beginning of the next delta cycle or to a specified delta cycle.	`-delta`
Run to the beginning of the next phase of the simulation cycle.	`-phase`
Run until the beginning of the next scheduled digital process or to the beginning of the next delta cycle, whichever comes first.	`-process`
Run until the analog solver hands simulation control to the digital solver. This point is considered a synchronization point.	`-sync`

See also "Starting or Resuming a Simulation" on page 463.

Syntax

```
run
    -clean
    -delta [cycle_spec]
    -next
    -phase
    -process
    -return
    -step
    [-timepoint] [time_spec] [-absolute | -relative]
    -sync
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-clean`	Runs the simulation to the next point at which it is possible to create a checkpoint snapshot with the `save -simulation` command.
	`-delta` [*`cycle_spec`*]	
		Runs the simulation for the specified number of delta cycles. If *`cycle_spec`* is omitted, runs the simulation to the beginning of the next delta cycle.
	`-next`	Runs one line of source code, stepping over any subprogram calls.
		If the current execution point is a VHDL non-zero `wait` statement, `run -next` might behave the same as `run -step`. For example, if the current execution point is a `wait` statement, which suspends the current process, another process might be scheduled to run at the current simulation time. In this situation, `run -next` 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 `wait`, set a line breakpoint on the line and enter a `run` command.
	`-phase`	Runs to the beginning of the next phase of the digital simulation cycle. The two phases of a simulation cycle are signal evaluation and process execution.

Modifiers	Options and Arguments	Function
	`-process`	Runs until the beginning of the next scheduled digital process or to the beginning of the next delta cycle, whichever comes first. In VHDL, a process is a `process` statement. In Verilog-AMS it is an `always` block, an `initial` block, or some other behavior that can be scheduled to run. **Note:** For the purposes of `run -process`, the `analog` block is not considered a process.
	`-return`	Runs until the current subprogram (task, function, procedure) returns.

Modifiers	Options and Arguments	Function
	`[-timepoint]` `[`*`time_spec`*`]` `[-absolute` | `-relative]`	

		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 `deltas` as the unit. For example,
		
		`ncsim> run 10 deltas`
		
		This is the same as `run -delta 10`.
		
		If you include a time specification, the simulator stops at the specified time with the digital solver active.
		
		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 `run` command does not stop the simulation at 500 ns.
		
		`ncsim> stop -object x`
		`Created stop 1`
		`ncsim> run 500 ns`
		`Stop 1 {x = 0} at 10 ns`
		`ncsim> run`
		
		`run -timepoint` without *`time_spec`* runs the simulation until the next scheduled analog or digital event.

	`-step`	Runs one behavioral statement, stepping into subprogram calls. **Note:** 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.
	`-sync`	Runs until the analog solver next hands control to the digital solver.

Examples

The following command runs the simulation until an interrupt occurs or until simulation completes.

```
ncsim> run
```

The following command advances the simulation to 500 ns absolute time. The `-timepoint` option is not required.

```
ncsim> run -timepoint 500 ns -absolute
```

The following command advances the simulation 500 ns relative time. With a time specification, `-relative` is the default.

```
ncsim> run 500 ns
```

The following command runs one behavioral statement, stepping into any subprogram calls.

```
ncsim> run -step
```

The following command runs one behavioral statement, stepping over any subprogram calls.

```
ncsim> run -next
```

The following command runs until the current subprogram returns. The subprogram can be a task, function, or procedure.

```
ncsim> run -return
```

The following two commands are equivalent. They both run the simulation for 5 delta cycles.

```
ncsim> run -delta 5
ncsim> run 5 deltas
```

The following command runs the simulation until the digital solver next becomes active.

```
ncsim> run -sync
Ran until 2 NS + 0
```

save

Creates a snapshot of the current simulation state. You can then use the `restart` command to load the saved snapshot and resume simulation. (For information about using the save and restart feature in SimVision, see the "Saving, Restarting, Resetting, and Reinvoking a Simulation" section, in chapter 3 of *SimVision User Guide*.)

Important

If you are using the UltraSim solver, you can use the `save` command but not the `restart` command.

You must specify a snapshot name with the `save` command. The snapshot name can be specified using [*`lib`*.]cell[:*`view`*] 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 `worklib.top:rtl`,

`save ckpt1`	saves `worklib.ckpt1:rtl`
`save top:ckpt1`	saves `worklib.top:ckpt1`
`save otherlib.top`	saves `otherlib.top:rtl`
`save :ckpt1`	saves `worklib.top:ckpt1`

The snapshot name must be a simple name containing only letters, numbers, and underscores.

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.

Note: 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,

```
ncsim> save -simulation -overwrite snap1
```

- Save snapshots to a separate library. For example,

  ```
  mkdir INCA_libs/snaplib
  ncsim -f ncsim.args
  ncsim> run 1000 ns
  ncsim> save -simulation snaplib.snap1
  ncsim> run 1000 ns
  ncsim> save -simulation snaplib.snap2
  ```

- Remove snapshots using the *ncrm* utility. For example,

  ```
  ncrm -snapshot worklib.snap1
  ```

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 `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 restore the environment by executing this script with the Tcl `source` command, or you can use the `-input` option when you start the simulator.

For example:

```
ncsim> save :ckpt1

ncsim> save -environment ckpt1.tcl

ncsim> restart :ckpt1

ncsim> source ckpt1.tcl
```

or

```
ncsim> -tcl cell:ckpt1 -input ckpt1.tcl)
```

Note: 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.

See “Saving, Restarting, Resetting, and Reinvoking a Simulation” in the “Simulating Your Design With ncsim” chapter of the `Simulating Your Design` book.

Syntax

```
save [-simulation] snapshot_name [-overwrite]

save -environment [filename]

save -commands [filename]
```

save Command Modifiers and Options

Modifiers	Options and Arguments	Function
	`-commands` [*filename*]	`save -commands` is the same as `save -environment`.
	`-environment` [*filename*]	Create a Tcl script that captures the current breakpoints, databases, probes, aliases, and predefined Tcl variable values. The *filename* argument is optional. If no file name is specified, the script is written to standard output.
	[`-simulation`] *snapshot_name*	Create 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.
	`-overwrite`	Overwrites an existing snapshot.

Examples

- How to Save a Snapshot of the Current Simulation State on page 580
- How to Use the save -environment Command on page 581
- How to Use save, restart, and analog on page 582

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.

```
ncsim> save -simulation :ckpt1
```

The following command saves the simulation state in *lib*.top:ckpt1.

```
ncsim> 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.

```
ncsim> 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

```
ncsim -tcl hardrive
ncsim: v2.1.(p1): (c) Copyright 1995 - 2003 Cadence
Design Systems
Loading snapshot worklib.hardrive:module .... Done
```

Start the simulator.

```
ncsim> stop -create -line 32
Created stop 1
ncsim> stop -create -object hardrive.clk
Created stop 2
ncsim> probe -create -shm hardrive.data
Created default SHM database ncsim.shm
Created probe 1
```

Set a line breakpoint, an object breakpoint, and create a probe. The probe command creates a default SHM database.

```
ncsim> run
0 FS + 0 (stop 2: hardrive.clk = 0)
./hardrive.v:13    clk = 0;

ncsim> run
50 NS + 0 (stop 2: hardrive.clk = 1)
./hardrive.v:16 always #50 clk = ~clk;

ncsim> save -environment env1.env
```

Save the debug settings in a file called `env1.env`.

```
ncsim> more env1.env
```

Look at the file contents.

```
set assert_report_level {note}
set assert_stop_level {error}
set autoscope {yes}
set display_unit {auto}
set tcl_prompt1 {puts -nonewline "ncsim> "}
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 ncsim.shm ncsim.shm -default
probe -create -name 1 -database ncsim.shm hardrive.data
scope -set hardrive

ncsim> exit
```

Exit from the simulator.

```
foghorn% ncsim -tcl hardrive
ncsim: v1.2.(b9): (c) Copyright 1995 - 2000 Cadence
Design Systems
Loading snapshot worklib.hardrive:module
.................... Done
```

Restart the simulator.

Commands, *continued*

Notes, *continued*

```
ncsim> stop -show
No stops set
```

The new session has no breakpoints.

```
ncsim> source env1.env
```

Source the `env1.env` file.

```
ncsim> stop -show
1 Enabled Line: ./hardrive.v:32 (scope: hardrive)
2 Enabled Object hardrive.clk
ncsim> probe -show
1 Enabled hardrive.data (database: ncsim.shm) -shm
ncsim> database -show
ncsim.shm Enabled (file: ncsim.shm) (SHM) (default)
```

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

```
ncsim> database -open waves -into waves.shm -default
Created default SHM database waves
```

Open a default SHM database called `waves`.

```
ncsim> probe -create -all
Created probe 1
```

Probe all signals in the current scope.

```
ncsim> stop -create -time -absolute 200ns
Created stop 1
```

Create a breakpoint at absolute time 200ns.

```
ncsim> run
200 NS + 0 (stop 1)
```

Run until 200ns.

```
ncsim> analog -show
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.

Commands, *continued*

Notes, *continued*

```
ncsim> save :ckpt1
Saved snapshot amslib.top:ckpt1
```

Save the first snapshot

```
ncsim> analog -stop 20 us
```

Lengthen the simulation.

```
ncsim> analog -reltol 5e-03
```

Change the `reltol` value.

```
ncsim> analog -show
vabstol = 1.000000e-06 (alterable)
iabstol = 1.000000e-12 (alterable)
reltol = 5.000000e-03 (alterable)
stop = 20 us (alterable)
```

Examine the new values.

```
ncsim> stop -create -time -absolute 400ns
Created stop 2
```

Create another breakpoint.

```
ncsim> run
400 NS + 0 (stop 2)
```

Run until 400ns.

```
ncsim> save :ckpt2
Saved snapshot amslib.top:ckpt2
```

Save the second snapshot.

```
ncsim> restart :ckpt1
Loaded snapshot amslib.top:ckpt1
```

Reload the first snapshot.

```
ncsim> analog -show
vabstol = 1.000000e-06 (alterable)
iabstol = 1.000000e-12 (alterable)
reltol = 1.000000e-03 (alterable)
stop = 14 us (alterable)
```

Check that tolerance values and stop time are the same values you had originally...

```
ncsim> time
200 NS
```

...and that you are back at 200ns in the simulation.

```
ncsim> restart :ckpt2
Loaded snapshot amslib.top:ckpt2
```

Now reload the second snapshot.

```
ncsim> analog -show
vabstol = 1.000000e-06 (alterable)
iabstol = 1.000000e-12 (alterable)
reltol = 5.000000e-03 (alterable)
stop = 20 us (alterable)
```

Check for the changed `reltol` value and the lengthened simulation.

```
ncsim> time
400 NS
```

Run to 400ns, again!

```
ncsim> run 200 NS
Ran until 600 NS + 0
```

Run another 200ns.

```
ncsim> restart :ckpt1
Loaded snapshot amslib.top:ckpt1
```

Switch back to the first snapshot.

```
ncsim> time
200 NS
```

Check where you are.

Commands, *continued*

Notes, *continued*

```
ncsim> 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)
```

Run until analog stop time.

```
ncsim> analog -stop 100us
```

At analog stop time, you can still lengthen the simulation.

```
ncsim> 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
```

Run until analog stop time.

```
ncsim> restart :ckpt1
Loaded snapshot amslib.top:ckpt1
```

At analog stop time, you can still restart a snapshot.

```
ncsim> 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.
```

Exit and complete the 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`)

Note: 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
```

Modifiers and Options

Modifiers	Options and Arguments	Function
`-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`	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.
	`-all`	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.

Modifiers	Options and Arguments	Function
`-describe`	[*`scope_name`*]	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 `scope -describe` 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. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.
	`-names`	Displays only the names of each declared item in the scope.
	`-sort` {`name` | `kind` | `declaration`}	Specifies the sort order. There are three possible arguments to the `-sort` 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

Modifiers	Options and Arguments	Function
`-drivers`	[*scope_name*]	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 `scope -drivers` 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. See "Enabling Read, Write, or Connectivity Access to Digital Simulation Objects" on page 443 for details on specifying access to simulation objects.
`-disciplines`	[scope_name]	Lists all resolved net disciplines within the given scope, or within the current debug scope if no scope is given.
	`-recurse`	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.
	`-all`	Lists all resolved net disciplines in all top-level scopes. If used with `-recurse`, recursively lists all resolved net disciplines in the entire design.
	`-sort` `name` | `kind` | `declaration`	Sort the nets alphabetically by net name, by discipline (`electrical`, `logic`, etc.) or by the order they are declared in the source code. The default is to sort by discipline.

Modifiers	Options and Arguments	Function
`-list`	[*`line`* | *`start_line`* *`end_line`*] [*`scope_name`*]	Prints lines of source code for the specified scope, or for the current debug scope if no scope is specified. You can follow the `-list` 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 *`start_line`* or *`end_line`*.
`-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.
`-show`		Shows scope information, including the current debug scope, instances within the debug scope, and top-level modules in the currently loaded model.

Example

The following example prints the name of the current scope. The `-set` modifier is not required.

```
ncsim> scope -set
```

The following example sets the debug scope to scope `u1`. The `-set` modifier is not required.

```
ncsim> scope -set u1
```

The following example moves the debug scope up one level in the hierarchy.

```
ncsim> 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

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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).

```
ncsim> 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).

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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`.

```
ncsim> 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 (`ncelab -access +r+w`). In this example, only the drivers for wires and registers with read access are shown.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> scope -list
```

The following example lists the source for scope `u1`.

```
ncsim> scope -list u1
```

The following example displays line 12 of the source for the current debug scope.

```
ncsim> scope -list 12
```

The following example lists lines 10 through 15 of the source for the current debug scope.

```
ncsim> 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.

```
ncsim> scope -list - 10
```

The following command lists lines of source for the current debug scope, beginning with line 30.

```
ncsim> 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.

```
ncsim> 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
```

status

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.

Syntax

```
status
```

Modifiers and Options

None.

Example

The following example shows the type of statistics displayed by the `status` command.

```
ncsim> 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
         -condition {tcl_expression}
         -delta delta_cycle_number [-relative | -absolute]
              [-start delta_cycle_number]
              [-modulo delta_cycle_number]
         -line line_number
              { -unit unit_name | [scope_name] [-all] }
              [-file filename]
         -object object_names
         -process process_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

`-create`		Creates a breakpoint. This modifier must be followed by an option that specifies the breakpoint type:
		`-condition`
		`-delta` (VHDL only)
		`-line`
		`-object`
		`-process` (VHDL only)
		`-time`

Modifiers	Options and Arguments	Function
	`-condition {`*`tcl_expression`*`}`	Sets a breakpoint that triggers when any digital object referenced in *`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). *`tcl_expression`* must contain at least one digital object. **Note:** 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” on page 611 for details on the format of conditional expressions. Objects included in a `-condition` 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” on page 443 for details.
	`-continue`	Resumes the simulation after executing the breakpoint. The simulator does not go into interactive mode.
	`-delbreak` *`count`*	Deletes the breakpoint after it has triggered *`count`* number of times.

Modifiers	Options and Arguments	Function
	`-delta` *`delta_cycle_num`* [`-absolute`] [`-relative`] [`-start` *`delta_cycle_num`*] [`-modulo` *`delta_cycle_num`*]	Sets a breakpoint that triggers when the simulation delta cycle count reaches the specified delta cycle. 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. See the "Setting a Delta Breakpoint" section of the "Debugging Your Design" chapter in *Cadence Verilog Simulation User Guide* for more information.

Modifiers	Options and Arguments	Function
	`-execute` *`command`*	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.
	`-if` {*`tcl_expression`*}	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. See "Tcl Expressions as Arguments" on page 611 for more information on the format of *`tcl_expression`*. Objects included in an `-if` 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" on page 443 for details on specifying access to simulation objects.

Modifiers	Options and Arguments	Function

	`-line` *`line_number`*
	
	{ `-unit` *`unit_name`* | [*`scope_name`*] [`-all`] }
	
	[`-file` *`filename`*]
		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 `-unit`. 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 *`scope_name`* method, use the `-all` option. If the scope name is omitted, then the current debug scope is used.

Modifiers	**Options and Arguments**	**Function**
		The `-file` 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 `-linedebug` option to enable the setting of line breakpoints. See the “Setting a Source Code Line Breakpoint” section of the “Debugging Your Design” chapter in *Cadence Verilog Simulation User Guide* for more information.
	`-name` *`break_name`*	Specifies a name for the breakpoint. This name can then be used to delete, disable, or enable the breakpoint. If you do not use `-name`, breakpoints are numbered sequentially.
	`-object` *`object_name`*	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 “Enabling Read, Write, or Connectivity Access to Digital Simulation Objects” on page 443 for details on specifying access to simulation objects. See the “Setting an Object Breakpoint” section of the “Debugging Your Design” chapter in *Cadence Verilog Simulation User Guide* for more information.

Modifiers	Options and Arguments	Function
	`-process` *`process_name`*	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 `-linedebug` to enable the setting of process breakpoints. See the "Setting a Process Breakpoint" section of the "Debugging Your Design" chapter in *Cadence Verilog Simulation User Guide* for more information.
	`-silent`	Suppresses the display of the message that is printed when a breakpoint triggers.
	`-skip` *`count`*	Tells the simulator to ignore the breakpoint for the first *`count`* times that it triggers. You can use `-skip` to set a breakpoint on the `n`th occurrence of an event; in particular, you can use it to get inside `for` loops.

	`-time` *`time_spec`*
	`[-absolute]`
	`[-relative]`
	`[-start` *`time_spec`*`]`
	`[-modulo` *`time_spec`*`]`
		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.
		
		Note: The digital solver is always active when the simulator stops for a time breakpoint.
		
		Use `-start` 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 `-start`.

Modifiers	Options and Arguments	Function
		The `-modulo` option is similar to `-start`. Use `-modulo` to specify the absolute simulation time of the first stop time for a repeating stop. This differs from `-start` 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 `-modulo 250`, the simulation stops at time 50, 150, 250, and so on. When you execute a `save -environment` 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 *Cadence Verilog Simulation User Guide* 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 *Cadence Verilog Simulation User Guide* 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 *Cadence Verilog Simulation User Guide* for more information.

Modifiers	Options and Arguments	Function
`-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 *Cadence Verilog Simulation User Guide* 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 *Cadence Verilog Simulation User Guide* 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, `ncsim` assigns a sequential number as the name. This breakpoint is called `1`.

```
ncsim> stop -create -object sum
Created stop 1
```

The following command creates a breakpoint named `mybreak` that stops simulation when `sum` changes value.

```
ncsim> stop -object sum -name mybreak
Created stop mybreak
```

The following command creates a breakpoint that triggers when `sum` changes value. The breakpoint is ignored the first 3 times it triggers.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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" on page 611 for details on the syntax of the argument to the `-if` option.

```
ncsim> 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 -nounit] > 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.

```
ncsim> stop -object clk
ncsim: *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.

```
ncsim> stop -object compSig
ncsim: *W,STALOB: Cannot set stop on analog object:
         top.dac.compSig. This object ignored.
ncsim: *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.

```
ncsim> stop -line 10
```

The following command creates a breakpoint that stops simulation when line number 13 in scope `counter` is about to execute.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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:

```
ncsim> 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`.

```
ncsim> 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.

```
ncsim> 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.

```
ncsim> 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" on page 611 for more information on writing these expressions.

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: ncsim> stop -condition {[value %d top.count] = 100}
VHDL: ncsim> 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:

```
ncsim> 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:

```
ncsim> 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.

```
ncsim> stop -condition {#%dcount = 100}
```

For Verilog, you can use the standard notation (for example `4'b0011`). For example, you can set the breakpoint on `count` as follows:

```
ncsim> stop -condition {#count = 32'd100}
ncsim> stop -condition {#count = 32'b00000000000000000000000001100100}
```

VHDL does not have the same type of notation. Vectors must be enclosed in quotation marks, as shown in the next example.

```
ncsim> stop -condition {#count = "00000000000000000000000001100100"}
```

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.

```
Verilog: ncsim> stop -condition {#count[0] == 1}
VHDL: ncsim> stop -condition {#count(0) == '1'}
```

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.

```
Verilog: ncsim> stop -condition {[value %b count[0]] == 1'b1}
VHDL: ncsim> stop -condition {[value %b count(0)] == '1'}
```

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.

```
Verilog: ncsim> stop -condition {#clock == 1} -if {#en1 || #en2}
VHDL: ncsim> stop -condition {#clk_n == '1'}
                   -if {#enable=='1'|| #reset_n=='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.

```
ncsim> stop -time 5 ns -start 5 ns
    -execute {if {#clk == ’0’} {force clk ’1’}
    else {force clk ’0’}} -continue
```

Process Breakpoints

The following command sets a breakpoint that stops the simulation whenever the process called `:load_action` is executed.

```
ncsim> 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.

```
ncsim> stop -line 12 -name break1
Created stop break1
ncsim> stop -object data -name break2
Created stop break2
ncsim> stop -show
break1  Enabled         Line: ./shortdrive.v:12 (scope: top)
break2  Enabled         Object top.data
ncsim>
```

In the following command sequence, breakpoint `break1` is first disabled with the `-disable` modifier and then enabled with the `-enable` modifier.

```
ncsim> stop -show
break1  Enabled         Line: ./shortdrive.v:12 (scope: top)
break2  Enabled         Object top.data
ncsim> stop -disable break1
ncsim> stop -show
break1  Disabled        Line: ./shortdrive.v:12 (scope: top)
break2  Enabled         Object top.data
ncsim> stop -enable break1
ncsim>
```

The following command deletes breakpoint `break1`.

```
ncsim> stop -delete break1
```

To disable, enable, or delete the two breakpoints `break1` and `break2`, any of the following commands could be used.

```
ncsim> stop -delete *1 *2
```

```
ncsim> stop -delete break?
```

```
ncsim> stop -delete br*
```

The following command displays information on any breakpoint beginning with `v` or `b`.

```
ncsim> 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*}

  Note: 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'bx`, 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.

  ```
  ncsim> stop -condition #w[1] == 1
  ncsim> 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.

  ```
  ncsim> stop -time 100 ns -if {r[1] == 1}
  ```

  Use the following commands:

  ```
  Verilog:
  ncsim> stop -time 100 ns -if {[value r[1]] == 1'b1}
  ncsim> stop -time 100 ns -if {#r[1] == 1}
  VHDL:
  ncsim> stop -time 100 ns -if {[value r(1)] == '1'}
  ```

```
    ncsim> 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:
ncsim> stop -condition {[value %d out] = 12}
ncsim> stop -condition {#%dout = 12}
VHDL:
ncsim> stop -condition {[value %d out] = 12}
ncsim> stop -condition {#%dout = 1}
```

For VHDL, you must enclose vectors in quotation marks and single-bit entities in single quotation marks. For example,

```
Verilog: ncsim> stop -condition {#clock == 1}
VHDL: ncsim> stop -condition {#clock == ‘1’}

Verilog: ncsim> stop -condition {#count = 4’b0101}
VHDL: ncsim> 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.

time

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

If no unit is given, the value of the `$display_unit` variable is used. This variable is set to `auto` by default.

Syntax

```
time [[10 | 100]time_unit | auto | module]
    -delta
    -nounit
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-delta`	Includes the delta cycle count. **Note:** 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.

Examples

```
ncsim -tcl board
ncsim: v1.0.(p2): (c) Copyright 1995 - 2000 Cadence Design Systems
ncsim> run 100 ns
5  count= X, f=x, af=x
Ran until 100 NS + 0
```

The following command displays the current simulation time in ns.

```
ncsim> time ns
100 NS
```

The following command displays the current simulation time in fs.

```
ncsim> time fs
100000000 FS
```

The following command displays the current simulation time in 100 times the base unit of fs.

```
ncsim> time 100fs
1000000 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.

```
ncsim> time fs
100000000 FS
```

```
ncsim> time auto
100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope.

```
ncsim> time module
100 NS
```

The following command displays the current simulation time using the timescale of the current debug scope and including the delta cycle count.

```
ncsim> time module -delta
100 NS + 0
```

The following command displays the current simulation time with no time unit.

```
ncsim> time -nounit
100
```

value

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.

For information about using `value` with unnamed branches, see "Specifying Unnamed Branch Objects" on page 620.

The `value` command is supported on user-defined net types. For more information on user-defined net types, refer to User-Defined Net Type and Resolution Function on page 262

Syntax

```
value [format ...] object_name ...
    -potential
    -flow
```

Modifiers and Options

Modifiers	Options and Arguments	Function
	`-potential`	Returns the potential of analog branches that follow on the command line. This option is ignored for any other kind of object.
	`-flow`	Returns the flow value of analog branches that follow on the command line. Returns the flow value of analog objects that have existing Tcl current probes. This option is ignored for any other kind of object.

For Verilog, the valid formats are

`%c`	character
`%s`	string

`%b`	binary
`%d`	decimal
`%o`	octal
`%x`	unsigned 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 defaults are used:

- analog—`%g`
- time—`%d`
- integer—`%d`
- real—`%g`
- reg—`$vlog_format`
- wire—`$vlog_format`

where `$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_vector and std_logic_vector values. Otherwise, the format specifier is ignored for VHDL values.

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. For more information, see the "Driver-Receiver Segregation" section of the "Mixed-Signal Aspects of Verilog-AMS" chapter, in the *Cadence Verilog-AMS Language Reference*.

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.

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:

```
ncsim> 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

```
ncsim> 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.

```
ncsim> value data
4'h2
ncsim> value %o data
4'o02
ncsim> value %b data
4'b0010
ncsim> value %d data
4'd2
ncsim> value %g data
2
ncsim> value %f data
2.000000
ncsim> value %e data
2.000000e+00

ncsim> value %b data %d q
4'b0010 4'd1
ncsim> 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.

```
ncsim> value clk
ncsim: *E,RDACRQ: Object does not have read access: hardrive.clk.
```

where

Displays the current location of the simulation. This includes the current simulation time and the current scope.

Syntax

```
where
```

Modifiers and Options

None.

Example

```
ncsim> where
TIME: 3400 NS + 0
Scope is (board.counter)
ncsim>
```

Specifying Unnamed Branch Objects

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:

```
ncsim> 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:

```
ncsim> describe a_b
a_b........analog net (electrical ) = 0
ncsim> 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.

C

Source Protection

Virtuoso 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 the commands you use and in the implementation details.

Language	Method for Protection
Verilog-A, Verilog-AMS, Verilog (digital), VHDL-AMS, and VHDL (digital) code	`ncprotect`
Spectre code	`spectre_encrypt`

Note: You cannot probe any nets that go *through* an encrypted design unit.

You can use source protection with the following AMS Designer simulator and analog solver configurations:

- Spectre solver using the simulation front end (SFE) parser
- UltraSim solver using the SFE parser

Using ncprotect

When you use the `ncprotect` 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

   For information about inserting the protection pragmas in your source code, see "Using the Protection Pragmas" on page 623.

2. Run the `ncprotect` command on the input files containing the regions to be protected.

   This command creates a new source file in which the regions marked for protection are 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.

Using the Protection Pragmas

You use the protection pragmas to mark regions for protection in Verilog-AMS, Verilog (digital), VHDL-AMS, and VHDL (digital) code and Verilog-A code in your model files.

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.

Note: The following tasks do nothing when they are located inside an area that is protected: `$strobe`, `$fstrobe`, `$display`, `$fdisplay`, `$debug`, `$fdebug`, `$write`, `$fwrite`.

The following two examples show how to use the `protect begin` and `protect end` pragmas in a source file. The first 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 end
endmodule
```

This next 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 next example illustrates how to protect a region in a VHDL description:

```
architecture behavior of myblock is
-- pragma protect
-- pragma 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 end
END myblock;
```

The ncprotect Command

The `pragma protect`, `protect begin`, and `protect end` pragmas mark the regions you want to protect; encryption actually occurs when you run the `ncprotect` command on the source description files. The syntax of the `ncprotect` command is as follows:

```
ncprotect [-options] hdl_source_file [hdl_source_file ...]
[-APpend_log]
[-AUtoprotect]
[-Extension output_file_extension]
[-File filename]
[-Help]
[-LAnguage {vlog | vhdl}]
[-LOgfile logfile_name]
[-Messages]
[-NOCopyright]
[-NOLog]
[-NOStdout]
[-Overwrite]
[-Version]
```

For complete information, and many examples, see "ncprotect" in the *Protecting IP Source Files* book.

Processing a source description with the `ncprotect` 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.

The protected version of the first example in the previous section 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
hjQ2rsuJMpL9F3O43Xx7zf656dz2xxBxdnHC0GvJFJG3Y5HL0dSoPcLMN5Zy6Iq+
ySMMWcOGkowbtoHVjNn3UdcZFD6NFlWHJpb7KIc8Php8iT1uEZmtwTgDSy64yqLL
SCaqKffWXhnJ5n/936szbTSvc8vs2ILJYG4FnjIZeYARwKjbofvTgA==
//pragma protect end_key_block
//pragma protect digest_block
uilUH9+52Dwx1U6ajpWVBgZque4=
//pragma protect end_digest_block
//pragma protect data_block
jGZcQn3lBzXvF2kCXy+abmSjUdOfUzPOp7g7dfEzgN96O2ZRQP4aN7kqJOCA9shI
jcvO6pnBhjaTNlxUJBSbBA==
//pragma protect end_data_block
//pragma protect digest_block
tzEpxTPg7KWB9yMYYlqfoVE3lVk=
//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 `ncprotect`, 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 `ncprotect` 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`.

```
ncprotect src.v
```

The following command specifies an extension `myext` for the protected version of `design.v`: `design.v.myext`.

```
ncprotect design.v -extension myext
```

Note: If the name of the protected file conflicts with the name of an existing file, the `ncprotect` command does not create the protected file; instead, it issues a message that alerts you to the conflict.

Protecting All Modules in a Source Description

The `ncprotect -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) protects all modules in the specified source file automatically. 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 `ncprotect -autoprotect` command ignores them.

This option is particularly useful for protecting libraries that contain a large number of files with many modules.

Using spectre_encrypt

Using `spectre_encrypt`, you can encrypt Spectre model files.

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.

   For information about inserting the keywords in your model file, see “Using the protect and unprotect Keywords” on page 627.
4. Save the netlist.
5. Run `spectre_encrypt` on the input files containing the regions to be protected.

Note: `irun` can process design files that contain models encrypted using `spectre_encrypt` (when subcircuit boundaries are not encrypted). `irun` cannot process subcircuits or SPICE-in-the-middle constructs encrypted using `spectre_encrypt` because of the encrypted boundaries.

Using the protect and unprotect Keywords

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 *Virtuoso Spectre Circuit Simulator User Guide*.

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. The syntax of the `spectre_encrypt` command is given below, but for complete information, see “Encryption” in the *Virtuoso Spectre Circuit Simulator User Guide*.

```
spectre_encrypt [-i input_file] [-o output_file][-all]
```

where

input_file	The path and name of the file to be encrypted. If you do not specify the input file, the standard input is encrypted.
output_file	The path and name of a file to hold the encrypted model. The extension that you use for *`output_file`* must be the same extension used on *`input_file`*. If you do not specify the output file, the encrypted model is displayed as standard output in the terminal window.
-all	Encrypts the entire Spectre model file, ignoring any `protect` and `unprotect` 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.

Protection Guidelines

Be aware of the following behaviors as you use source protection:

- Verilog-AMS Protection on page 629
- Protection Guidelines for Automatically Inserted Connect Modules on page 629
- Forced Use of CMI 3.0 on page 629

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.

Obsolete Approach for Source Protection

This section describes an obsolete method of protecting source code, which Cadence supports for compatibility with existing designs. With this approach, you can

- Protect selected modules or regions within modules
- Automatically protect all modules

Protecting Selected Regions in a Source Description

To protect the source description of selected modules or regions, follow these steps:

1. Place two compiler directives in the source description to define the protected region: `` `protect `` marks the beginning of the protected region; `` `endprotect `` marks the end of the protected region.
2. Create a copy, with a different name, of the source description file.

   This file serves as a back-up.
3. Protect the original Verilog-AMS source description file with the command

   ```
   verilog orig_source_file +protect
   ```

   This command creates a new source file in which the regions marked for protection are unreadable. By default, the new file has the same name as the original file, but with an appended `p`.
4. Rename the newly created file, giving it the name of the original source file.

   This step makes the protected modules available to the AMS software.

The ‘protect and ‘endprotect Compiler Directives

You can use the compiler directives `` `protect `` and `` `endprotect `` inside or outside a module, provided that you pair each `` `protect `` directive with an `` `endprotect `` directive in the same source file. If you insert a `` `protect `` directive without a corresponding `` `endprotect `` directive, the compilation appears successful. However, when you recompile the protected source file, an error occurs.

You can use multiple sets of the `` `protect `` and `` `endprotect `` directives within modules. However, you cannot nest blocks of source code bounded by `` `protect `` and `` `endprotect `` directives inside one another.

The following two examples show how to use the `` `protect `` and `` `endprotect `` compiler directives in a source file. In the first example, the module `top_design` contains a region that is marked for protection.

```
module top_design (a, b, c)
    bottom inst ();
    `protect
        initial
        $display ("Inside module top_design");
    `endprotect
endmodule
```

In the next example, the entire module, including the module name, is marked for protection.

```
`protect
    module bottom ();
        initial
            begin
                $display ("Inside module bottom");
            end
        endmodule
`endprotect
```

The +protect Command-Line Option

Although the `` `protect `` and `` `endprotect `` directives mark the regions to be protected in your source description, the protection actually occurs when you protect the source file with the `verilog +protect` command. For example, the following example uses the `verilog +protect` command to protect several Verilog-AMS source files:

```
verilog src4.v src5.v src6.v +protect
```

Compiling a source description with the `verilog +protect` command protects only the regions marked with `` `protect `` and `` `endprotect `` compiler directives. The compiler creates a new source file that differs from the original file in the following ways:

- The directives `` `protect `` and `` `endprotect `` become `` `protected `` and `` `endprotected `` respectively.
- The regions marked for protection in the original source description become unreadable.

After the files in the previous two examples are compiled for protection, they take the following forms.

```
module top_design (a, b, c);
    bottom inst ();
    `protected
  a*lejodi)dlj@lsfj4gRekv*9l#sIjnd<;pXywUHvow%emhiITvne(@mengTVpe
  prK58s53<gf:dneURtnd&8ejsWqpsu*ehtsY=wkxOrkp$
    `endprotected
endmodule
```

```
`protected
  fkeop*456gjkl@%^&^&s85Kfmv(:wjvdwLSchrmx*2uPQjsu=:wucgwigIWsuxnt
  pr"W84&@(shxjMvn02:wjd8%&!0s$
`endprotected
```

The new, protected source file does not overwrite the original, unprotected source file. When you compile the original source file with `verilog +protect`, you can specify an optional file extension to be automatically appended to the name of the protected source file. If you do not specify an extension, the `verilog +protect` command automatically appends a `p` to the name of the protected file.

For example, the following command line protects the file `src.v`. Because no extension is specified, the command produces a protected file called `src.vp`.

```
verilog src.v +protect
```

The following command line specifies an extension `.myext` to be appended to the file `design.v`. As a result, the `verilog` command generates a protected source file called `design.v.myext`.

```
verilog design.v +protect.myext
```

Note: If the name of the protected file conflicts with the name of an existing file, the `verilog` command does not create the protected file; instead, it issues a message that alerts you to the conflict.

Protecting All Modules in a Source Description

The `verilog +autoprotect` command protects all modules in the specified source file automatically. You do not need to insert the `` `protect `` and `` `endprotect `` compiler directives in any source description that you plan to compile with `+autoprotect`. If these directives already exist in your source file, the `verilog +autoprotect` command ignores them

This option is particularly useful for protecting libraries that contain a large number of files with many modules. Compiling a source file with `verilog +autoprotect` creates a new source file that differs from the original source file in the following ways:

- The directive `` `protected `` is inserted after all module names, immediately before their port and terminal lists.
- The source descriptions inside all modules become unreadable.
- The `` `endprotected `` directive is inserted just before the `endmodule` keyword in modules.

To see how compiling with `+autoprotect` alters a source file, consider the following source code:

```
module top_design (a, b, c);
    input a,b;
    output c;
    initial
        $display ("Inside module top_design");
endmodule
```

Compiling the previous module example with `+autoprotect` creates the protected file shown in the following example:

```
module top_design 'protected a%jioDT:3e(prlXCWN67suwOpw%3(j&ls)?l
j8wPQhsmchALsxy23XM#&0):3Wbv
9DwoPs,x>s:2yTJfSlsBx,>?uri839tkd%whfx8$
'endprotected endmodule
```

Notice that the module name and the keywords `module` and `endmodule` remain outside the protected region. Anything following the module name—typically from the port list onward—is protected. Protection ends prior to the keyword `endmodule` to make it easier to scan files during library searches.

The new, protected source file does not overwrite the original, unprotected source file. When you compile the original source file with `verilog +autoprotect`, you can specify an optional file extension to be automatically appended to the name of the protected source file. If you do not specify an extension, the `verilog +autoprotect` command automatically appends a `p` to the name of the protected file.

For example, the following command line protects the file `design.v`.

```
verilog design.v +autoprotect.protall
```

Because the extension `.protall` is specified, this command produces a file called `design.v.protall`, in which all modules are protected.

The next example uses `+autoprotect` to protect multiple Verilog-AMS source files.

```
verilog src1.v src2.v src3.v +autoprotect
```

Because no extension is specified, the default extension is used and the files of protected modules are called `src1.vp`, `src2.vp`, and `src3.vp`.

Note: If the name of a protected file conflicts with the name of an existing file, the `verilog +autoprotect` command does not create the protected file; instead, it issues a message that alerts you to the conflict.

D

Using the Profiler

The profiler measures where CPU time is spent during simulation, and it provides information that you can use to modify your designs for better simulation.

To run the profiler, use the `-profile` option when you run the AMS Designer simulator (`ncsim`). For more information about the `ncsim` command, see "ncsim Command Syntax and Options" on page 449.

When the simulator exits, the profiler writes the profile to a file called `ncprof.out` 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:

`%hits`	The percentage of the total hits detected in the activity
`#hits`	The raw number of hits detected in the activity
`%cost`	The percentage of simulator time-steps required to service the analog operator or expression.
`domain`	The solver, either analog or digital
`#inst`	The number of instances
`name`	The name or description of the activity
`type`	One of the following types: `dVar` (a digital variable that causes a digital to analog event), `cross` (an @cross event), `filter` (a transition)
`instance`	The full hierarchical name of an instance and the line number where it is defined

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*. For more information about interpreting the digital results, see Chapter 14, in *Cadence Verilog Simulation User Guide*.

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 example shows 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 *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)
```

E

Migrating to an amsd Block from prop.cfg

If you are migrating from using a `prop.cfg` file to using an amsd block, the following information can help you to map `prop.cfg` statements to `amsd` block statements.

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

`string prop sourcefile "`*`file`*`"`	`include "`*`file`*`"`
	`amsd {…}`

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

The `amsd` block does not support the `inst` keyword (`prop.cfg`) directly. Instead, you can use occurrence-based instance paths. For example, instead of

```
inst I0 {…}
```

consider

```
path top.I0 {…}
```

for which the `amsd` block equivalent is

```
portmap subckt=bar_subckt
config inst=top.I0 use=spice
```

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` **Note:** 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

For a stub instance, consider the following `prop.cfg` syntax:

```
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"; }
```

The `amsd` block equivalent syntax is:

```
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
}
```

default Translations

You can specify default cell and bus settings using the `config` and `portmap` statements:

prop.cfg Syntax	amsd Block Syntax
`default hdl_cell="nand2"`	`config cell=nand2 use=hdl`

prop.cfg Syntax	amsd Block Syntax
`default sourcefile_opts_common= "-auto_bus -bus_delim []"`	`portmap autobus=yes busdelim="[]"`

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
`string prop hdl_cell="tb"`	`config cell=tb use=hdl`
`string prop hdl_cell="cell1 cell2"`	`config cell="cell1 cell2" use=hdl`

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

`inst xana_top2 sim_stub="-cell analog_top -src analog_top.scs"`	`portmap stub=analog_top match=spice`
	`config inst=xana_top2 use=stub`

`path (top).xana_top2 sim_stub= "worklib.analog_top:module"`	`portmap stub=worklib.analog_top:module match=verilog`
	`config inst=top.xana_top4.I0 use=stub`

See also “Stub Instance Translations” on page 641.

string prop sourcefile_opts Translations

The `sourcefile_opts` properties (`prop.cfg`) translate to `amsd` block syntax as follows:

prop.cfg Syntax	amsd Block Syntax
`sourcefile_opts="-auto_bus"`	`portmap autobus=yes`, or no `autobus` specification (`autobus=yes` is the default)
`sourcefile_opts="-bus_delim <>"`	`portmap busdelim="<>"`
`sourcefile_opts="-cell_case keep"`	`portmap ... cellcase=keep`
`sourcefile_opts="-case_map upper"`	`portmap casemap=upper`

`sourcefile_opts=`
`"-exclude_bus=itunea"`
	`portmap excludebus=itunea`

`sourcefile_opts=`
`"-in_port in1*in1"`
	`portmap input="in1*in1"`

`sourcefile_opts="-input mica"`
or
`sourcefile_opts="-input ti"`
	No special entry in `amsd` block. Instead, use
	`-amsi mica:file_mica.cir`
	or
	`-amsi ti:file_ti.cir`
	on the `ncelab` on 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"`
	`refformat=verilog`

`sim_mode = "s"`	`*ultrasim .usim_opt sim_mode="s"` **Note:** We do not support this ability now.

`speed=2`	`*ultrasim .usim_opt speed=2` **Note:** We do not support this ability now.
`Escaped names: path (\1_wrap ).I1`	`config inst="top.I4.1_I1"` Use the actual unescaped name. **Note:** This is still an open area.

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 became 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 MMSIM. 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 Virtuoso® 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

NC

Abbreviation for *Native Compiled*.

This acronym is usually used now to specify a product of the Incisive family. For example, NC-VHDL, NC-Sim, and NC-Verilog are products included with Incisive, while ncvhdl, ncvlog, ncelab, ncsim, and irun are specific tools within these products.

Natively compiled code and can also be known as Native Code.

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.

ncelab

Native code elaborator. The elaborator gathers various portions of a design and creates a snapshot that can be simulated (with ncsim). See the "Elaborating the Design With ncelab" chapter of the *Elaboration Command-Line Options* book for more information.

ncsim

Native code simulator. See the "Simulating Your Design With ncsim" chapter of the *Simulating Your Design* book for more information.

ncvhdl

Native code VHDL compiler. See "Compiling VHDL Source Files with ncvhdl" for more information.

ncvlog

Native code Verilog compiler. See the "Compiling Verilog Source Files with ncvlog" chapter of the *Compiling Verilog Source Files* book for more information.

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 irun, ncvlog, ncvhdl, and ncelab 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 Virtuoso® 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 `ncsim` 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.

AIUM

Abbreviation for *AMS Designer Incisive Use Model*.

AIUM 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 Incisive 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 AIUM 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.

Index