Notice

The information contained in this document is subject to change without notice.

Agilent Technologies makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Agilent Technologies shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms that apply to this software product is available upon request from your Agilent Technologies representative.

Restricted Rights Legend

Use, duplication or disclosure by the U. S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 for DoD agencies, and subparagraphs (c) (1) and (c) (2) of the Commercial Computer Software Restricted Rights clause at FAR 52.227-19 for other agencies.

Agilent Technologies
395 Page Mill Road
Palo Alto, CA 94304 U.S.A.

Copyright © 1998-2004, Agilent Technologies. All Rights Reserved.

Acknowledgments

Mentor Graphics is a trademark of Mentor Graphics Corporation in the U.S. and other countries.

Microsoft®, Windows®, MS Windows®, Windows NT®, and MS-DOS® are U.S. registered trademarks of Microsoft Corporation.

Pentium® is a U.S. registered trademark of Intel Corporation.

PostScript® and Acrobat® are trademarks of Adobe Systems Incorporated.

UNIX® is a registered trademark of the Open Group.

Java™ is a U.S. trademark of Sun Microsystems, Inc.
## Contents

1 **Getting Started**
   - Simulating Verilog-A Views in RFDE .................................................. 1-1
   - Accessing Verilog-A Based Example Devices ................................... 1-2
   - Loading Verilog-A Examples ............................................................... 1-3
   - Using a Verilog-A Device in a Simulation ........................................ 1-3
   - Modifying a Verilog-A Device ............................................................... 1-8
   - Licensing .............................................................................................. 1-10

2 **Using Verilog-A with ADSsim in RFDE**
   - Loading Verilog-A Modules ................................................................. 2-1
     - The Auto Loading Mechanism ......................................................... 2-2
     - Explicit Loading of Verilog-A Modules in the Netlist ................. 2-2
   - Overriding Built-in Devices ................................................................. 2-3
   - Using Models with Verilog-A Devices ................................................ 2-3
   - The Verilog-A Compiled Model Library Cache ............................... 2-5
   - Controlling the Auto Compilation Process ....................................... 2-6
   - Verilog-A Operator Limitations in Harmonic Balance and Circuit Envelope .... 2-7
   - Parameters .......................................................................................... 2-8
   - Hierarchy and Module Resolution ...................................................... 2-8
   - Modifying the Simulator's Model Search Path ................................... 2-9
   - The Compiler Include Search Path ...................................................... 2-9
   - Interaction with the Loading of Dynamically Linked UCMs .................... 2-10

3 **Introduction to Model Development in Verilog-A**
   - Creating a Linear Resistor in Verilog-A ............................................. 3-1
     - Adding Noise to the Verilog-A Resistor .................................... 3-2
   - Creating a Linear Capacitor and Inductor in Verilog-A ......... 3-3
   - Creating a Nonlinear Diode in Verilog-A ........................................ 3-4
     - Adding an Internal Node to the Diode ...................................... 3-5
     - Adding Noise to the Diode ........................................................... 3-6
     - Adding Limiting to the Diode for Better Convergence .............. 3-6
   - Using Parameter Ranges to Restrict Verilog-A Parameter Values .... 3-7
   - Creating Sources in Verilog-A .......................................................... 3-7
   - Creating Behavioral Models in Verilog-A ....................................... 3-8
   - Using Hierarchy to Manage Model Complexity .............................. 3-11

A **Verilog-A in Process Design Kits**

B **Compilation Tools**
   - Compiling Large Files on HP-UX ....................................................... B-1
Index
Chapter 1: Getting Started

Verilog-A for RF Design Environment (RFDE) brings support for the Verilog-A language into RFDE. The system has been designed to work seamlessly in the Cadence Virtuoso® Analog Design Environment and enable the direct use of existing Verilog-A based libraries without modification.

This documentation describes how to use the system and also provides a brief overview of the Verilog-A language. You can utilize this information to understand how to use Verilog-A modules with the ADS Analog RF Simulator (ADSSim). For a detailed discussion of the Verilog-A language, refer to the Agilent Technologies Verilog-A Reference Manual.

RFDE Verilog-A leverages existing Virtuoso Verilog-A support. If you are not familiar with Verilog-A in Virtuoso, refer to the Cadence Verilog-A Reference Manual.

Simulating Verilog-A Views in RFDE

When a Verilog-A cell view is simulated for the first time, the RFDE Verilog-A compiler compiles the associated Verilog-A code and caches the result so that it may be used directly in all subsequent simulations of that view.

A typical simulation status showing the cache creation message and a compilation / load message is shown below.

HPEESOF SIM (*) 2003C.day Dec 14 2003 (built: 12/14/03 21:28:57)

A Verilog-A compiled model library (CML) cache has been created at
'/users/bobl/hpeesof/agilent-model-cache/cml/0.99/hpux11'

Compiling Verilog-A file
'/rfdeExamples/VerilogA_Tutorial/sinRampSrc/veriloga/veriloga.va'

AGILENT_VACOMP (*) 2003C.day Dec 14 2003 (built: 12/15/03 01:49:33)
Tiburon Design Automation (R) Verilog-A Compiler Version 0.99.121203.
Copyright (C) Tiburon Design Automation, Inc. 2002-2003. All rights reserved.

Loading Verilog-A module 'sinRampSrc' from
'/users/bobl/hpeesof/agilent-model-cache/cml/0.99/hpux11/veriloga_09106/lib.hpux11/veriloga.cml'.

TRAN Tran1[1] <input.ckt>  time=(0 s->100 ms)
Resource usage:
  Total stopwatch time: 23.21 seconds.
Getting Started

**Note**  There are several Verilog-A specific facilities in Virtuoso that may be used as aids in developing Verilog-A cell views. The most important of these is the Model Writer facility and the Verilog-A debugger. These tools can be used in the Virtuoso environment when developing Verilog-A modules, which may later be simulated in RFDE.

---

**Accessing Verilog-A Based Example Devices**

A variety of Verilog-A compact models are provided as examples when the product is installed. To access the Verilog-A based devices provided as an example library, the library path must be added to your cds.lib file. To add the library path to your cds.lib file, from the Cadence Command Interpreter Window (CIW),

1. Choose **Tools > Library Manager**. The Cadence Library Manager form appears.

2. Choose **Edit > Library Path** from the Library Manager. The Cadence Library Path Editor (CdsLibEditor) form appears. A list of available libraries and their associated paths are displayed in the CdsLibEditor.

3. Click the last empty Library cell in the CdsLibEditor and enter `compactLib` as the Library name.

4. Click the Path cell next to the Library cell and enter the path for the library. In this case, the library is located in the `$HPEESOF_DIR/tiburon-da/rfde/libs` directory.

<table>
<thead>
<tr>
<th>Library</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>compactLib</td>
<td><code>$HPEESOF_DIR/tiburon-da/rfde/libs/compactLib</code></td>
</tr>
</tbody>
</table>

5. Choose **File > Save**. The new library definition is added to your cds.lib file and the Library Manager window refreshes to include your new library.

For more information on the Cadence Library Path Editor, refer to your Cadence Library Path Editor User Guide.

**Note**  Installation of Verilog-A devices is automatic once the library is defined in your cds.lib file.

---

1-2  Accessing Verilog-A Based Example Devices
Loading Verilog-A Examples

Once the Verilog-A product is installed, several Verilog-A examples, including the Verilog-A Tutorial, are made available in the list of RFDE examples. Verilog-A examples can be loaded and opened the same way as any other RFDE example. To load and open the Verilog-A Tutorial example from the Cadence CIW,

1. Choose **Tools > RFDE Examples > Load Example**.
2. Scroll down the list and click the **VerilogA_Tutorial** example.
3. After the example has loaded, choose **Tools > RFDE Examples > Open Example** and click **VerilogA_Tutorial**. A group of example libraries is listed and available to choose from.

4. Click one of the Top-level cells in the example and then click **OK** to open the example.

For more information on working with examples, refer to your RFDE Examples documentation.

Using a Verilog-A Device in a Simulation

Verilog-A devices provide many of the capabilities as well as the look and feel of a traditional, built-in device, with the added benefit that the end-user can choose to modify the underlying equations. Verilog-A also enables you to share models between the ADSsim and Spectre simulators. The use model for Verilog-A in RFDE is similar to the Verilog-A use model in Spectre, so Spectre users should find using Verilog-A
models in the RFDE very familiar. To illustrate this, a popular GaAs FET model is simulated in RFDE and modified using the procedure described below.

1. Open the tutorial_PSFETV example from the VerilogA_Tutorial. Choose Tools > RFDE Examples > Open Example, select the VerilogA_Tutorial and choose tutorial_PSFETV.

   **Note** The steps used to create the tutorial_PSFETV example are listed below.

2. An instance of the PSFET (Parker-Skellern) device from the compactLib library has been used in this design, renamed as psfetv.

3. From the analogLib library, two instances of vdc were added, one for the gate, and the other for the drain.

4. Also from the analogLib library, the gnd components near the two voltage sources were added, as well as the one near the source.

5. A wire was inserted from the positive connection of the first voltage source to the gate of the Parker-Skellern FET, another wire from the other voltage source to the drain, and finally, to each of the appropriate gnd components. Your circuit should now appear similar to Figure 1-1.
6. Set up your simulation.

7. Add design variables \( V_d \) and \( V_g \) to represent the value of \( V_{dc} \) for the drain and gate sources, respectively. Set up a DC sweep of the drain from 0 to 5 volts as shown in Figure 1-3.
Getting Started

Figure 1-3. Analog Design Environment Setup

8. Select the **Tools > Parameter Sweep** menu item to step the gate voltage from -2 to 0 volts in 5 steps as shown in **Figure 1-4** and then click the Simulate and Run command button.
9. The simulator will then run and the Data Display window will open. Insert a rectangular plot and then insert the V0.i data trace, which represents the drain current. Because of the direction of the source, a minus sign is necessary to invert the trace. At this point the IV curve should look similar to the results shown in Figure 1-5.
Getting Started

Figure 1-5. Results of V0.i Data Trace

Modifying a Verilog-A Device

A powerful feature of Verilog-A is that a user can make modifications to the equations that describe the behavior of the device and these changes can be available in the simulator, with no loss of analysis functionality. It is expected that models will be distributed with their source code with the understanding that end-users will modify the equations for any number of reasons. For example, the user may want the equations to better reflect some aspect of their device behavior, or they may want to delete code that is not necessary to describe their device behavior.

Verilog-A for RFDE includes a set of example models implemented in Verilog-A and available to users as conventional models. These should be considered proof-of-concept models only, not built-in models ready for design. The source code is available for some of these models. In this example, the Parker-Skellern model will be modified and re-simulated.

Open the cellview tutorial_PSFETV in the library VerilogA_Tutorial. This cell view contains a copy of the Parker-Skellern model. It has been re-implemented as psfetv to avoid any unintentional problems when the file is modified as part of this exercise.

The source code for the model is found in the file:

$HPEESOF_DIR/veriloga_tutorials/psfetv/veriloga/veriloga.va
The user can choose to edit the file with a conventional text editor from the operating system, or can edit the model from within the Analog Design Environment. The latter method will automatically invoke the Spectre Verilog-A interpreter, which will parse the file and report errors. During simulation, the ADSsim simulator will automatically check to see if the file is out-of-date relative to the Compiled Model Library (CML) file and will recompile if necessary. The Spectre Verilog-A interpreter and the Verilog-A compiler in ADSsim are not the same program and may issue warnings differently. While the Verilog-A compiler in ADSsim supports the Spectre interpretation of the Verilog-A language, depending on the version of the language supported, each may report errors differently.

1. Follow the same steps as described in “Using a Verilog-A Device in a Simulation” on page 1-3 and verify that the results are the same as in the previous case.

2. Using any text editor, open the Verilog-A file. It is also possible to use the Descent -> Edit command; however, this will invoke the Spectre interpreter when the file is closed.

3. Find the equation that describes the drain current, \( I_d \):

\[
I_d = A \cdot B_T \cdot (1 + \lambda \cdot V_{dst}) \cdot (V_{gt}^q - (V_{gt} - V_{dt})^q)
\]

Purely for demonstration purposes, change the power law relation from \( q \) to \( q/2 \) in the first power term:

\[
I_d = A \cdot B_T \cdot (1 + \lambda \cdot V_{dst}) \cdot (V_{gt}^{q/2} - (V_{gt} - V_{dt})^q)
\]

4. Save the file and start a simulation. The program will detect that the file has changed and will compile the source file.

If there had been a compile error, the Status window would display the error and the associated line number.

The Data Display window shown in Figure 1-6 now shows very different results from those in Figure 1-5, demonstrating that the modification to the equation did indeed take effect.
Licensing

There is a single license associated with both the compilation of Verilog-A modules, and the loading of those modules into the ADSsim simulator. That license is called: sim_veriloga

There are two conditions under which the system pulls this license,

- **Compilation**: Once for each Verilog-A file compiled. A Verilog-A file is compiled if the file is in the Verilog-A model path or if it is loaded via the #load command.

- **Loading**: When a compiled Verilog-A module is loaded in the system by either a #load or the auto-loading mechanisms, the system pulls a license on the first load. Note that if this Verilog-A file has just been compiled, the license has already been pulled. Verilog-A files that over-ride built-in devices pull a license even if they are not used.
Chapter 2: Using Verilog-A with ADSsim in RFDE

Once Verilog-A modules are loaded into the simulation environment, they may be used just like any other device in the system. The ADS Analog RF Simulator (ADSsim) has a Verilog-A model search path. Any Verilog-A module in the search path is available to be instantiated in the netlist or used on the schematic. This Verilog-A module loading mechanism is called auto loading as the search path is only traversed and modules are only loaded when the simulator encounters an unknown device definition. The alternative loading mechanism is called explicit loading. In this case, the system always loads a particular Verilog-A module whether the simulator needs it or not. These loading mechanisms complement each other and will be explained in detail here.

The \#load command is the ADSsim equivalent of the Spectre ahdll_include command and may be used in ADSsim in much the same way as ahdll_include is used in Spectre. Once a module is loaded into the system, it takes on the look and feel of a built-in ADSsim device. The module name becomes the device definition name, the module parameters become device parameters, and, uniquely to Verilog-A, the module may have an optional model card.

During the loading process, a Verilog-A file is compiled into a shared library. These libraries are referred to as Compiled Model Libraries (CML). In general, users do not need to be concerned about the compilation process. The process has been designed to enable users to focus on writing and using Verilog-A modules rather than on the mechanics of compilation and CML file management. The system uses a CML cache. Verilog-A files are compiled once and stored for later use in the CML cache. The system manages the CML files, updating them only when necessary. As a user, you will see Verilog-A files being compiled the first time you use the files and subsequently only when you modify the files.

Loading Verilog-A Modules

In RFDE, you will generally not be directly concerned with Verilog-A device loading. When a Verilog-A cell is netlisted, the netlisting mechanisms also netlist the appropriate \#load line command, which in turn loads that particular Verilog-A module into the system. When Verilog-A fragments are used in a Process Design Kit (PDK), the PDK developer will also have chosen the appropriate loading scheme and again you, as an end user, will not need to understand the mechanisms involved.
The Auto Loading Mechanism

Verilog-A files (files with a .va or .vams extension) that reside in the simulator’s Verilog-A model search path are automatically compiled and loaded by the simulator, on demand, when an unknown device is encountered in the netlist. The Verilog-A model search path has four components,

__INCLUDE__

$HOME/hpeesof/veriloga
$HPEESOF_DIR/custom/veriloga
$HPEESOF_DIR/veriloga

When searching for an unknown device, the system will sequentially look for a matching Verilog-A module name in each of the above directories. The first module found is loaded and a status message issued informing you that a Verilog-A module has been loaded from a particular file in the model search path.

The __INCLUDE__ variable is set from the value of Include Path in the Simulation Files Setup form (Setup > Simulation Files). When you have veriloga as a particular directory, you can use this setting to include that directory in the Verilog-A model path.

Placing a Verilog-A module in your local $HOME/hpeesof/veriloga directory will make it visible to all simulations for this user account. The site custom directory ($HPEESOF_DIR/custom/veriloga) should be used to make Verilog-A modules available to all users of a particular RFDE installation. Finally the site directory ($HPEESOF_DIR/veriloga) is generally only used by Agilent Technologies to deliver new Verilog-A modules as part of the RFDE system.

Each file in any directory in the path must have a unique set of module names. In addition the module namespace across all files in a given directory must be unique. Having duplicate Verilog-A module names across multiple files in a directory is flagged as an error. The system cannot determine which module to use. To correct the error, modify the module names so that they are unique and re-run the simulation.

Explicit Loading of Verilog-A Modules in the Netlist

Explicit loading refers to explicitly loading all modules in a named Verilog-A file using the #load command. As mentioned earlier, in RFDE, as a Verilog-A view user or a PDK user you do not need to understand loading. The system automatically inserts a #load command into the netlist when a Verilog-A view is instantiated. #load is a preprocessor directive with the following syntax,
When filename is an absolute name the system directly loads that name. When filename is relative the system searches each directory in the Verilog-A model search path and loads the first file that matches that name. Note that all modules in a file are loaded (unlike auto loading where only the required module is loaded into the system).

#load is a preprocessor command and as such it is processed early in the simulation boot phase. #load commands can be placed anywhere in the netlist, but since it is a preprocessor command, there is no regard for network hierarchy. #load processing occurs before auto loading so when a device is explicitly loaded, the auto loading mechanisms are never invoked even if the device is in the search path.

**Overriding Built-in Devices**

When a device has the same name as a built-in device, and that Verilog-A device is in the Verilog-A model search path or is explicitly loaded, then the Verilog-A device overrides the built-in device. For example, if we place a file containing a module called R in a Verilog-A file in the model search path, then the module R will override the built-in R (the linear resistor). When a built-in device is overridden, the system issues a status message warning of the override similar to the one shown below.

```
Loading Verilog-A module 'R' from
'/users/bobl/hpeesof/agilent-model-cache/cml/0.99/hpux11/veriloga_02387_2
0031215_135130_031762/lib.hpux11/resv.cml'.
This module overrides the built-in 'R' (Linear Two Terminal Resistor).
```

In this example, the built-in R (linear resistor) is being overridden.

**Note** Overriding built-ins is a powerful and convenient way to use Verilog-A based devices in place of an existing device, but it should be used with care. The system does not check that the device being loaded has the same parameter list as the built-in device that it is overriding. Without careful attention to detail, you can easily cause netlist syntax errors relating to unmatched parameters.

**Using Models with Verilog-A Devices**

The model card is a simulator convenience that enables you to share device parameters among multiple device instances. Verilog-A devices are unique in that
they have a flexible relationship with the model card. Each Verilog-A module loaded into the system introduces a new device and a new model - both of the same name.

All parameters on a Verilog-A device may either be specified on the model, on the instance, or in both places. When a Verilog-A device references a model, it first reads all of the model parameters. Parameters specified directly on the instance then override the model settings. This behavior is particularly convenient when two devices share almost the same model. In this situation, they may reference a single model and override the parameters that are particular to the instances themselves.

For example, let us take the diode module (PNDIODE) from the tutorial project. The module may be instantiated directly as,

```
PNDIODE:d1 1 0 Is=3e-14 Rs=1 N=1.1 CJ0=1e-12 TT=0
```

or it may be used more conventionally with a model card as,

```
Model PNDIODE Dmod1 IS=3e-14 RS=1 N=1.1 CJ0=1e-12 TT=0
Dmod1:d1
Dmod1:d2
Dmod1:d3
```

The decision to use a model or not is your choice. Generally, models are convenient when there are two or more instances that have the same parameter sets, as above where d1, d2, and d3 all share the same model. We can go one step further in Verilog-A when almost all model parameters are the same. For the PNDIODE, let us assume that RS is the only parameter that varies from instance to instance. We use Dmod1 again but simply override RS on each instance that requires an RS different from the model

```
Dmod1:d1 RS=2
Dmod1:d2 RS=3
Dmod1:d3 RS=4
Dmod1:d4
```

In this example, d1 through d3 override RS while d4 inherits its value from the model, Dmod1. This model parameter inheritance with instance parameter override is ideally suited to mismatch modeling where two devices share almost the same model except for one or two parameters.

While this behavior is powerful, it leads to some complications in any analysis that involves an operation that modifies a parameter. Operations that modify parameters are used in sweeping, tuning, optimization, and statistical analyses. We will use sweeping in this explanation, but the rules outlined here apply to the other analyses in exactly the same way.
Device instance parameter sweeping operates in the usual way. To sweep a Verilog-A instance parameter, simply reference the parameter in the normal way in the sweep dialog box. To sweep a model parameter, reference the model parameter, again in the same way as is done for any model.

When a model parameter is swept, only those instances that inherit the parameter are affected. Instances that reference the model but override the swept model parameter are not affected. When all instances override the model parameter, you will see no change in results as a function of the sweep variable. In the above example if $RS$ on DMod1 was being swept, then the only instance affected by the sweep would be d4.

**Note** Only those model parameters that are actually specified on the model card may be swept. This is a model only limitation.

### The Verilog-A Compiled Model Library Cache

When Verilog-A modules are compiled, they are stored in a directory cache, which by default is created at,

```
$HOME/hpeesof/agilent-model-cache/
```

This cache is created the first time a Verilog-A file is compiled and entries are added or updated as Verilog-A files are compiled. An existing cache may be moved to a new location, but cache contents may not be modified in any way.

A cache is user, platform, and release-version specific. However, no cache locking mechanisms are used so you should not run two copies of ADSim on the same platform and attempt to share the same model cache. This will result in cache corruption. No cache recovery tools are provided in this release. If two sessions do interfere with the cache, then you must delete the cache and restart the simulation. The cache will then progressively rebuild itself as simulations involving Verilog-A modules are performed.

If two copies of ADSim must be run concurrently (perhaps one from the RFDE user interface and another in batch mode), then use a separate cache. The cache location may be modified by using the environment variable,

```
AGILENT_MODEL_CACHE
```
or a hpeesofsim.cfg configuration variable of the same name. The cache directory is created when the first Verilog-A file is compiled and you will see a status message similar to the following,

HPEESOFSIM (*) 2003C.day Dec 14 2003 (built: 12/14/03 21:28:57)

A Verilog-A compiled model library (CML) cache has been created at
'/users/bobl/hpeesof/agilent-model-cache/cml/0.99/hpux11'
Compiling Verilog-A file
' rfdeExamples/VerilogA_Tutorial/sinRampSrc/veriloga/veriloga.va'
AGILENT-VACOMP (*) 2003C.day Dec 14 2003 (built: 12/15/03 01:49:33)
Tiburon Design Automation (R) Verilog-A Compiler Version 0.99.121203.
Copyright (C) Tiburon Design Automation, Inc. 2002-2003. All rights reserved.

Loading Verilog-A module 'sinRampSrc' from
'/users/bobl/hpeesof/agilent-model-cache/cml/0.99/hpux11/veriloga_09106/lib.hpux11/veriloga.cml'.

TRAN Tran1[1] <input.ckt>   time=(0 s->100 ms)
Resource usage:
   Total stopwatch time: 23.21 seconds.

CML files vary in size depending on the size of the Verilog-A file and the platform used. It is recommended that you locate the cache on a local disk with at least 200M bytes of free space. A local cache will improve compilation and load times. You may delete the cache at any time, though you should never need to do so, unless the size of the cache has become unreasonable or when it contains many old CML files that are no longer used. Deleting the cache leads only to recompiles and its clean reconstruction. Do this only when there are no simulations running.

**Controlling the Auto Compilation Process**

By default, when a Verilog-A module is compiled, the CML file is placed in the model cache to be retrieved later. If you do not want to use the cache, you have the option of storing the CML files in platform specific directories under the Verilog-A source directory. This and other auto compilation features are controlled by a file called vamake.spec (Verilog-A make specifications file, hence the name). If a file with this name is in the Verilog-A source directory, then the system uses it. The file may contain two options,
USE_CACHE=
PREPARE_DIR=

The `USE_CACHE` variable may be set to `YES` or `NO`. It defaults to `YES`. When you set it to `NO`, then the cache is not used and CML files are written to the platform specific directories:

- `lib.hpx11`
- `lib.linux_x86`
- `lib.sun57`
- `lib.win32`

In some cases, you may want to compile the libraries and subsequently have the files loaded without any dependency checks. To do this, set the `PREPARE_DIR` variable to `NO`.

**Verilog-A Operator Limitations in Harmonic Balance and Circuit Envelope**

The Verilog-A language was first designed with time domain simulation algorithms in mind and, as such, has a set of features that are not easily supported in Harmonic Balance, Circuit Envelope analysis and their derivatives (LSSP, XDB, etc.). In these cases, we either approximate the behavior and/or issue an appropriate warning or error message.

Event-driven constructs such as cross and timer will not be triggered in Harmonic Balance analysis or in Circuit Envelope analysis. Since the cross event does not trigger in these analyses, it follows that the `last_crossing()` function is likewise not triggered. When these events are present in a Verilog-A module, the system issues a warning message saying they will not be triggered.

The `abstime()` function will return 0.0 in Harmonic Balance analysis. When used, the system issues a warning message. In this release, users should use built-in ADS source components to generate time-varying waveforms.

The `idtmod()` operator and Z-transform filters are not supported for Harmonic Balance or Circuit Envelope in this release. The transition and slew filters act as simple allpass elements. This latter behavior is consistent with the language reference specification for frequency domain analysis.

Modules which contain memory states are not compatible with Harmonic Balance. A module contains a memory state if it has local variables which retain values that are used at subsequent time steps.
Module and Parameter Naming

ADSsim and Verilog-A are case sensitive. Verilog-A also supports a set of special characters through an escaping mechanism. These special characters are not supported in this release and so all module names (and therefore ADSsim Verilog-A based device names) are alpha numeric with underscores and no leading number. The same is true of Verilog-A module parameters.

Parameters

Verilog-A supports reals, integers, arrays of reals, and arrays of integers. When an integer value is passed to a real parameter, it is converted to a real. When a real value is passed to an integer parameter, the real value is rounded to the nearest integer value and the system issues a warning indicating loss of precision in a parameter assignment. Ties are rounded away from zero.

The ADSsim functions list() and makearray() are two of several functions that may be used to generate arrays to be passed to Verilog-A array type module parameters. Verilog-A arrays have a single dimension; ADSsim arrays can be multi-dimensional. If a multi-dimensional array is passed, then an incorrect dimension error is issued.

As with scalar integer parameters, if a real array is assigned to an integer parameter array then a loss of precision warning message is issued and the system rounds each real in the array to the nearest integer value.

If either string or complex scalar or array values are passed to Verilog-A devices, then an unsupported type message is issued. No roundings or conversions are attempted.

Verilog-A parameters also optionally have a range specification. If a parameter value is specified that is outside the allowed range, then an out of range error message is issued and the system exits. Parameters can go out of range during a sweep, a tune, an optimization, or a statistical analysis. In each case, an out of range error is issued and the system exits. To avoid the problem, you must ensure that during parameterized analyses such as sweeps, Verilog-A parameter values are only varied across their allowed ranges.

Hierarchy and Module Resolution

Verilog-A allows you to describe components in terms of their structure or their behavior or a mixture of both. Frequently, Verilog-A modules in a module hierarchy are all contained in the same Verilog-A file. This is not a requirement and the fact
that the modules are in the same file implies no special module binding in terms of what modules actually get instantiated when a hierarchical module is itself instantiated in the ADSsim netlist.

When a Verilog-A module is instantiated, the system recourses down its hierarchy instantiating all of its children. When instantiating a child, the system uses the Verilog-A model search path. This means that the first child definition found in the path is instantiated, and that is not necessarily the child module defined in the same file as the parent.

**Modifying the Simulator’s Model Search Path**

The simulator defines its model search path in the simulator configuration file, $hpeesofsimsim.cfg$. To extend the search path, add the line,

```
USER_VERILOGA_MODEL_PATH=myDir1:myDir2:...
```

to your local configuration file. This directory or set of directories are pre-pended to the Verilog-A model search path and the system will then compile all Verilog-A modules and make them available to the Verilog-A auto loading mechanism.

In addition to the configuration parameter, an environment variable of the same name and the same syntax may be used. Use of the configuration file is preferred as its location determines the visibility of the additions.

**The Compiler Include Search Path**

When including files with a relative path via the Verilog-A `include` directive, the system first looks in the same directory as the Verilog-A file being compiled. If the file is not found in that directory, then the system looks in the Verilog-A system include directory given by,

```
HPEESOF_DIR/tiburon-da/veriloga/include
```

These are the only directories in the Verilog-A `include` search path in this release.
Interaction with the Loading of Dynamically Linked UCMs

The Verilog-A models and User-Compiled Models (UCM) using the dynamic linking mechanism share the same search path. When the system encounters an unknown device, it looks in each directory in the model search path for that device.

In each directory, it first looks to see if there is a Verilog-A module with a matching name in this directory. If there is, then this Verilog-A device is used and loaded. If no Verilog-A device is found, then the system looks to see if there is a UCM device in this directory. If a UCM is found, it is loaded. If no UCM is found, then the system moves on to the next directory in the model search path and does the same query until it finds the device or reaches the end of the path.

This means that in any given directory, a Verilog-A device takes precedence over a UCM device. But since Verilog-A and UCM devices are searched for in each directory, the Verilog-A modules later in the path do not hide UCM devices earlier in the path and vice-versa. When a Verilog-A device and a UCM device of the same name exist in the same directory, the Verilog-A device is loaded and no error or warning message is issued.

In normal usage, Verilog-A files reside in veriloga directories (described earlier) and UCM devices reside in other directories. When customizing the Verilog-A model path, you should avoid mixing Verilog-A and UCM devices in the same directory.
Chapter 3: Introduction to Model Development in Verilog-A

This chapter provides a brief introduction to the Verilog-A language by means of examples. A more complete description of the language is available in the Verilog-A Reference Manual. A simple resistor is first defined, then enhanced with noise. Models for capacitors and inductors are then developed. These models use the \texttt{ddt()} operator to automatically generate the time-dependent functionality of the devices. Finally, a nonlinear diode is created demonstrating modeling of more complex behavior.

Creating a Linear Resistor in Verilog-A

The linear resistor in the example code below was introduced in the previous chapter. This provides a simple example of the anatomy of a Verilog-A model. Line numbers are used here to help explain the code, but should not be included in the actual source code.

\begin{verbatim}
[1] `include "disciplines.vams"
[2] module R(p,n);
[3] electrical p,n;
[4] parameter real R=50.0;
[5] analog V(p,n) <+ R * I(p,n);
[6] endmodule
\end{verbatim}

Line [1] instructs the compiler to insert the contents of the file \texttt{disciplines.vams} into the text. This file contains the definitions that make the Verilog-A specific for electrical modeling.

Line [2] and line [6] declares the module block, within which the model behavior will be defined. The model is named \texttt{R} and has two ports, named “p” and “n”. Ports provide connections to other modules.

Line [3] declares that the ports \texttt{p} and \texttt{n} have the nature of those declared in the electrical discipline, as defined in the \texttt{disciplines.vams} header file. Natures and disciplines provide a way to map the general flows and potentials to particular domains, like electrical, thermal, or mechanical.

Line [4] declares one parameter, called \texttt{R}, and assigns it a default value of \texttt{50.0}. The default value is set if the simulator is not passed an assignment in the netlist. In this case, the parameter is explicitly declared as \texttt{real}. However, if this attribute (which
could also be integer) is not provided, the language infers the type from the default value. In this case, 50.0 would indicate a real type, whereas 50 would indicate an integer. The parameter declaration also includes a simple method to restrict the range values. This is described in “Using Parameter Ranges to Restrict Verilog-A Parameter Values” on page 3-7. Parameter values can not be modified by the Verilog-A code. If the value needs to be modified, it should be assigned to an intermediate variable.

The keyword analog in line [5] declares the analog block. In this case, it is a single statement. However, statements can be grouped together using begin/ end keywords to denote blocks which, in turn, can be named to allow local declarations. The simple, single statement includes several key aspects of the language. On the right hand side, the access function I(p,n) returns the current flowing from node p to n. This is multiplied by the value of the parameter R. The “<<” in line [5] is called the contribution operator and in this example contributes the value of the evaluated right hand side expression as the voltage from p to n.

**Adding Noise to the Verilog-A Resistor**

Verilog-A provides several ways to add noise, including frequency-independent, frequency-dependent, and piecewise linear frequency-dependent noise. In the case of a resistor, the thermal noise is \(4K*T/R\). The value of Boltzmann’s constant is available in another header file, constants.vams, as a macro definition. Verilog-A supports preprocessor commands similar to other languages like C. The details of macros are discussed in the Verilog-A Reference Manual, but in general, macros can be thought of as simple text substitutions, typically used to help make the code more readable and to gather all the constant definitions into one place. In the header file, the definition is,

```
#define P_K 1.3806226e-23
```

whereas in the Verilog-A code, the value is used as `P_K`. The temperature of the circuit is a value that can be changed outside of the model and so must be dynamically accessed. Verilog-A models use system functions to retrieve information that the simulator can change. The temperature environment parameter function is $temperature and returns the circuit’s ambient temperature in Kelvin.

The actual contribution of the noise is made with the white_noise() operator, which takes the noise contribution as an argument. Noise functions also allow for an optional string to label the noise contribution. Some simulators can sort the noise according to the labels.
Creating a Linear Capacitor and Inductor in Verilog-A

Capacitors and inductors are implemented in a similar way to resistors. However, these devices have dependencies on time. In the case of a capacitor, the relationship is,

\[
I = C \cdot \frac{dV}{dt}
\]

In this case, the contribution is a current through the branch. The right hand side includes a derivative with respect to time. This is implemented with the \texttt{ddt()} operator. The model then becomes,

\begin{verbatim}
`include "disciplines.vams"
module C(p,n);
inout p,n;
electrical p,n;
    parameter real C=0 from [0:inf);
analog I(p,n) <+ C * ddt(V(p,n));
endmodule
\end{verbatim}

This example also illustrates one use of the range functions in the parameter declaration. The "from [0:inf)" addition restricts the value of C from 0 up to, but not including, infinity.

Similarly, the inductor relationship is,

\[
V = L \cdot \frac{dI}{dt}
\]

and the source code is:

\begin{verbatim}
`include "disciplines.vams"
module L(p,n);
inout p,n;
electrical p,n;
    parameter real L=0 from [0:inf);
\end{verbatim}

Note that line [6] of the example code above shows the added noise.
Introduction to Model Development in Verilog-A

3-4 Creating a Nonlinear Diode in Verilog-A

Verilog-A is well-suited for describing nonlinear behavior. The basic PN junction diode behavior will be used as an example. The I-V relation is,

\[ I = I_s \left( \exp \left( \frac{V}{V_{th}} - R_s I \right) - 1 \right) \]

The implementation is shown below.

```verilog-a
`include "disciplines.vams"
`include "constants.vams"
module diode(anode, cathode);
electrical anode, cathode;
parameter real Area = 1.0 from (0:inf); //Area scaling factor
parameter real Is = 1e-14 from [0:inf]; //Saturation current [A]
parameter real Rs = 0.0 from [0:inf]; // Series resistance [Ohm]
parameter real N = 1.0 from (0:inf); //Ideality
parameter real Tt = 0.0 from [0:inf]; //Transit time [s]
parameter real Cjo = 0.0 from [0:inf]; //Junction capacitance [F]
parameter real Vj = 1.0 exclude 0; // Junction potential [v]
parameter real M = 0.5 from [0:inf]; //Grading coef
parameter real Fc = 0.5 from [0:1]; //Forward bias junct parm
parameter real Kf = 0.0; //Flicker noise coef
parameter real Af = 1.0 from (0:inf); //Flicker noise exponent
real Vd, Id, Qd;
real f1, f2, f3, Fcp;
analog begin
    f1 = (Vj/(1 - M)) * (1 - pow((1 - Fc), (1 - M)));
f2 = pow((1 - Fc), (1 + M));
f3 = 1 - Fc * (1 + M);
    Fcp = Fc * Vj;
    Vd = V(anode, cathode);
    // Intrinsic diode
    Id = Area * Is * \left( \exp \left( \frac{Vd}{(N * V_{th}) - R_s I(anode, cathode)} \right) - 1 \right);
    // Capacitance (junction and diffusion)
    if (Vd <= Fcp)
        Qd = Tt * Id + Area * Cjo * Vj * (1 - pow((1 - Vd / Vj), (1 - M)))/(1 - M);
    else
        Qd = Tt * Id + Area * Cjo * (f1 + (1 / f2) * (f3 * (Vd - Fcp) + (0.5 * M / Vj)) * (Vd * Vd - Fcp * Fcp)));
    I(anode, cathode) <+ Id + ddt(Qd);
endmodule
```
The more complicated behavior requires more complicated code. Comments are added to help clarify the source. Verilog-A supports two types of comment characters. Text to the right of // and text between /* and */ blocks will be ignored.

The analog block is extended from a single line to multiple lines using the begin and end keywords to indicate a compound expression. Intermediate variables are declared to make the code more readable. These variables are declared in the module but outside the analog block.

A new system function, $vt, is used. This function returns the thermal voltage calculated at an optional temperature. If no arguments are passed, the ambient circuit temperature is used. The mathematical operators exp() and pow() are also used. Verilog-A includes a wide range of mathematical functions.

### Adding an Internal Node to the Diode

Note that the transcendental diode relationship includes the drop in the junction voltage due to the series resistance. An alternate method of implementing the series resistance would be to add an internal node. An internal node (also called a net) is added by simply declaring the node as electrical, without adding the node to the port list on the module declaration line. The diode code changes as shown:

```verilog
`include "constants.vams"
`include "disciplines.vams"
module diode_va(anode, cathode);
    electrical anode, cathode, internal;
    parameter real Area = 1.0 from [0:inf];    //Area scaling factor
    parameter real Is = 1e-14 from [0:inf];   //Saturation current [A]
    parameter real Rs = 0.0 from [0:inf];     //Ohmic res [Ohm]
    parameter real N = 1.0 from [0:inf];      //Emission coef
    parameter real Tt = 0.0 from [0:inf];     //Transit time [s]
    parameter real Cjo = 0.0 from [0:inf];    //Junction capacitance [F]
    parameter real Vj = 1.0 exclude 0;        //Junction potential [V]
    parameter real M = 0.5 from [0:inf];      //Grading coef
    parameter real Kf = 0.0;                  //Flicker noise coef
    parameter real Af = 1.0 from [0:inf];     //Flicker noise exponent
    parameter real Fc = 0.5 from [0:1];       //Forward bias junct parm
    real Vd, Id, Qd;
    real f1, f2, f3, Fcp;
    analog begin
        f1 = (Vj/(1 - M))*(1 - pow((1 - Fc), 1 - M));
        f2 = pow((1 - Fc), (1 + M));
    end
endmodule
```
Introduction to Model Development in Verilog-A

```markdown
f3 = 1 - Fc * (1 + M);
Fcp = Fc * Vj;
Vd = V(anode, internal);
// Intrinsic diode
Id = Area * Is * ((Vd / (N * $vt)) - 1);
// Capacitance (junction and diffusion)
if (Vd <= Fcp)
  Qd = Tt * Id + Area * Cjo * Vj * (1 - pow((1 - Vd / Vj), (1 - M)))/(1 - M);
else
  Qd = Tt * Id + Area * Cjo * (f1 + (1 / f2) * (f3 * (Vd - Fcp) + (0.5 * M / Vj) * (Vd * Vd - Fcp * Fcp)));
I(anode, internal) <+ Id + ddt(Qd);
V(internal, cathode) <+ I(internal, cathode) * (Rs / Area);
end
endmodule

Adding Noise to the Diode

Noise is contributed in the same way as it was for the basic resistor. In this case, the shot noise equation shows the dependence on the diode current. The 1/f noise is added using the `flicker_noise()` operator, which takes as arguments the value of the noise contribution as well as the exponent to apply to the 1/f term.

```verilog
// Noise
I(anode, cathode) <+ white_noise(2 * 'P_Q * Id, "shot");
I(anode, cathode) <+ flicker_noise(Kf * pow(Id, Af), 1.0, "flicker");
```

The thermal noise from the series resistor is added in the same fashion as was done for the resistor. Note that the label is modified to indicate which resistor the noise is generated from. This is useful when the analysis supports Sort Noise by Name.

```verilog
// Series resistor
V(internal, cathode) <+ white_noise(4 * 'P_K * T * (Rs / Area), "Rs");
```

Adding Limiting to the Diode for Better Convergence

The exponential function used in the diode code can result in large swings in currents for small changes in voltage during the simulator’s attempts to solve the circuit equations. A special operator, `limexp()` can be used instead of `exp()` to allow the simulator algorithms to limit the exponential in simulator-specific ways. The specific algorithm used is simulator dependent.
Using Parameter Ranges to Restrict Verilog-A Parameter Values

The parameter declaration allows the range of the parameter to be conveniently restricted. At run time, the parameter value is checked to be sure it is acceptable. If it is not, the simulator issues an error and stops.

By default, parameters can range from -infinity to infinity. To restrict a range either the exclusive from (:) can be used, or the inclusive from [:] or a combination of the two. For example,

\[
\text{from (0 : 10)}
\]

will restrict the parameter from 0 to 10, excluding the value of 0 but including the value of 10.

Exceptions to ranges are indicated by the except attribute. For example,

\[
\text{except 5}
\]

will not allow the value of 5 to be passed.

Ranges and exceptions can be repeated and combined. For example,

\[
\text{parameter real X = 20.0 from (-inf: -10] from [10:inf);}
\]

can also be written as,

\[
\text{parameter real X = 20.0 exclude (-10:10);}
\]

If a simulator supports sweeping of parameters, the model developer will have to be aware of issues related to sweeping through ranges.

Creating Sources in Verilog-A

Analog sources can also be described with Verilog-A using the same concepts. Sources typically have some relation to the specific time during the simulation. The time is available from the $abstime function, which simply returns the real time (in seconds) of the simulation.

A simple sine wave source would have the form:

\[
\text{'include "disciplines.vams"}
\text{'include "constants.vams"
module sine_wave(n1,n2);
electrical n1,n2;
}
\]
Introduction to Model Development in Verilog-A

```verilog
parameter real gain = 1.0, freq = 1.0;
analog V(n1,n2) <+ gain * sin(2 * 'M_PI * freq * $abstime);
$bound_step(0.05/freq);
endmodule
```

The mathematical constant for PI is available as M_PI from the constants.vams header file. Note that the multiple parameter declarations were combined on one line as an alternative to declaring each on its own line.

The system function `$bound_step` restricts the simulator’s transient steps to the size 0.05/freq. This allows the model to define the resolution of the signal to be controlled.

An additional use of defining sources in Verilog-A is to create test bench circuits as part of the model source file. This test module would provide sources with appropriate values and sweep ranges to allow the validation of the model to be contained within the code definition. This is a useful method of providing portable tests when distributing models among different simulators.

Creating Behavioral Models in Verilog-A

Verilog-A enables the user to trade off between various levels of abstraction. Certain circuit blocks lend themselves to simple analog descriptions, resulting in improvements in simulator execution time compared to transistor level descriptions. Since Verilog-A supports all of the analysis functionality, the user is typically only trading off simulation accuracy when using a behavioral description of a circuit.

The Phase-Locked Loop (PLL) is a good example of a circuit that can be represented in behavioral blocks.

The Verilog-A source code below demonstrates a PLL circuit. The PLL consists of a phase detector, an amplifier, and a voltage controlled oscillator. In this example, a swept sine source is used to test the circuit.

The VCO and phase detector are defined as:

```verilog
// Voltage Controlled Oscillator
module vco(in, out);
inout in, out;
electrical in, out;
parameter real gain = 1, fc = 1;
analog V(out) <+ sin(2*M_PI*(fc*$abstime) + idt(gain*V(in)));
endmodule
```

3-8 Creating Behavioral Models in Verilog-A
// Phase Detector
module phaseDetector(lo, rf, if_);
    inout lo, rf, if_;
    electrical lo, rf, if_;
    parameter real gain=1;
    analog function real chopper;
        input sw, in;
        real sw, in;
        chopper = sw > 0 ? in : -in;
    endfunction // chopper
    analog V(if_) <+ gain*chopper(V(lo),V(rf));
endmodule

The modules use the keyword `inout` to declare that the ports are both input and output. Some simulators will check consistency of connection of ports (useful when ports are declared input or output only). ADSim will not.

The `phaseDetector` makes use of an analog function definition of chopper to simplify the code. Analog functions can be thought of as sub-routines that can take many values but return one value. This is in contrast to macros, which should be thought of as in-line text substitutions.

The PLL module uses hierarchy to instantiate these components:

// Phase Locked Loop
module pll(rf, out, ref, if_);
    inout rf, out, ref, if_;
    electrical rf, out, ref, if_;
    parameter real tau = 1m from (0:inf);
    parameter real loopGain = 1 from (0:inf);
    parameter real fc = 2.0k from (0:inf);
    real cap, res;
    electrical lo;
    phaseDetector #( .gain(2)) pd1(lo, rf, if_);
    vco #(.gain(loopGain/2), .fc(fc) ) vco1(out, lo);
    analog begin
        cap = 150e-9;
        res = tau / cap;
        V(out, if_) <+ I(out, if_)*res;
        I(out, ref) <+ ddt(cap*V(out,ref));
    end
endmodule

For testing, a swept frequency source is defined:

module sinRampSrc(p,n);
    inout p,n;
    electrical p,n;

3-10 Creating Behavioral Models in Verilog-A

Introduction to Model Development in Verilog-A

```verbatim
parameter real fStart = 1, hzPerSec = 1;
analog V(p,n) <+ sin(2*'$M_PI*'(fStart+hzPerSec/2*$abstime)*$abstime);
endmodule
```

The modules are connected in a circuit (using appropriate AEL and symbol definitions), and a transient simulation is run.

![Phase-Locked Loop Test Circuit](image)

**Figure 3-1. Phase-Locked Loop Test Circuit**

Plotting the Out node shows the VCO control voltage response to the swept frequency input, indicating the locking range of the circuit.
Verilog-A supports module hierarchy, which allows modules to be embedded in other modules, enabling you to create complex circuit topologies.

To use a hierarchy, the model developer creates textual definitions of individual modules in the usual fashion. Each module definition is independent, that is, the definitions can not be nested. Special statements within the module definitions instantiate copies of the other modules. Parameter values can be passed or modified by the hierarchical modules, providing a way to customize the behavior of instantiated modules.

The instantiation format is:

```module_name #(parameter list and values) instance name(port connections);```

For example, the previous definitions of R, L, and C can be used with a new module called RLC to create a simple filter.

```module RLC(in, out);
electrical in, out, n1, n2;
parameter real RR = 50.0 from [0:inf);
parameter real LL = 0.0 from [0:inf);
parameter real CC = 0.0 from [0:inf);
R #(R(RR)) R1(in, n1);
L #(L(LL)) L1(n1, n2);```
Introduction to Model Development in Verilog-A

C #(C(CC)) C1(n2, out);
endmodule

The RLC module creates a series R-L-C from the input port in to the output port out, using two internal nodes n1 and n2. The RLC module’s parameter values of RR, LL, and CC are passed to the modules R, L, and C’s parameters R, L, and C via #(R(RR)), #(L(LL)), and #(C(CC)).

A unique advantage of the Compiled Model Library file is that the Verilog-A source is effectively hidden from end users. This, coupled with Verilog-A’s hierarchical structure, gives model developers a simple way to distribute intellectual property without exposing proprietary information.
Appendix A: Verilog-A in Process Design Kits

Verilog-A for RFDE has been designed to support Process Design Kits (PDKs) and in particular to support the easy translation of Verilog-A content in an existing Cadence Spectre® based PDK.

In a typical SpectrePDK, Verilog-A modules are placed in the model file directory and the Spectre ahdl_include command is used in the PDK model file to make these Verilog-A modules available to the simulator.

In RFDE, the #load Command may be used in an analogous manner. A #load command is added to the RFDE PDK model file for each Verilog-A file in the PDK. For example, if a PDK contains,

```
PDK_DIR/model_file_directory/varactor.va
/resistor.va
```

then the PDF model file will contain,

```
... #load "veriloga", "varactor.va"
#load "veriloga", "resistor.va"
...
```

The model file directory is specified in the IncludePath of the Simulation Files Setup form and the model file is included in the system in the standard way. The Verilog-A modules are found in the model file directory because the include path above is always part of the #load search path.

This is all that is required if you want to ship the Verilog-A source with the PDK. You can also ship the Compiled Model Library (CML) files only if you need to protect model intellectual property that is otherwise visible in the Verilog-A source code.

To do this, first turn off CML cache usage by placing a Verilog-A make specifications file, `vamake.spec` in the model directory. Put a single line in this file,

```
USE_CACHE=NO
```

Now run any simulation that uses the PDK. You will see a platform specific directory under the model file directory called,

```
lib.arch
```

where arch is one of win32, linux_x86, hpux11, or sun57. This directory will contain the CML files, in this example,
Verilog-A in Process Design Kits

```
  varactor.cml
  resistor.cml
```

If all platforms are supported, then run the simulation on all platforms and there will be four directories `lib.win32`, `lib.hpux11`, `lib.sun57`, and `lib.linux_x86`. They will all contain CML files of the same names, but they will be platform specific.

You can now delete the contents of the Verilog-A files and add another line to `vamakesspec` as,

```
  USE_CACHE=NO
  PREPARE_DIR=NO
```

The latter line will instruct the compilation system not to check that the CML files are up to date, but simply load them directly.

For more information on developing RFDE PDKs, refer to the RFDE Cadence Library Integration documentation.
Appendix B: Compilation Tools

The Verilog-A compilation suite achieves a compile-complete solution by utilizing open source compilation tools on all platforms. The tools are not modified and supplied in compiled form only.

On Linux, HP-UX, and Sun, we use the gcc compiler version 3.2.3. For more information on gcc see gcc.gnu.org. On these platforms, the system supplied linker and assembler are used.

Compiled Model Library (CML) files created using these tools may be distributed without any licensing restrictions. The CML compilation is also designed to not require any shared libraries and so they may be easily moved from machine to machine of the same architecture.

All of these tools are installed within the system directory $HPEESOF_DIR/tiburon-da and are used only by the Verilog-A compilation system.

Compiling Large Files on HP-UX

The default C compiler used on HP-UX has some limitations with respect to Verilog-A files containing more than a few thousand lines of code. If you wish to compile such files on HP-UX, then you must use the cc compiler that is supplied as part of the aCC compiler tool set. Note that the Kernighan & Ritchie compiler that comes standard on all HP-UX installations is not sufficient and will not compile CML files.

To use cc, set the environment variable AGILENT_CML_BUILDER as,

```bash
export AGILENT_CML_BUILDER=agilent-cml_builder-cc.sh
```

Prior to booting RFDE, ksh shell syntax is shown here. You must also ensure that cc is in your path. With this setting, the system will use the native C compiler rather than the system supplied default compiler.
Compilation Tools
# Index

## Symbols
- `#load` command, 2-1, 2-2, A-1
- `$abstime` function, 2-7, 3-7
- `$bound_step` function, 3-8
- `$temperature` function, 3-2
- `$vt` function, 3-5
- `__INCLUDE__` (include path), 2-2
- `#include` compiler directive, 2-9
- `A-CC` compiler tool set, B-1
- ADSim, 1-1, 1-3, 1-10, 2-1, 2-5, 2-8, 3-9
- `ahdl_include` command, 2-1, A-1
- allpass elements, 2-7
- analog block, 3-2, 3-5
- analog descriptions, 3-8
- Analog Design Environment, 1-1
- analog keyword, 3-2
- analog sources, 3-7
- arrays, 2-8
- auto compilation features, 2-6
- auto loading, 2-1, 2-2, 2-3, 2-9
- auto loading
- Boltzmann constant, 3-2
- C compiler, B-1
- cache, 1-1
- case sensitivity, 2-8
- cc compiler, B-1
- CdsLibEditor, 1-2
- cell views, 1-2
- Circuit Envelope analysis, 2-7
- CML cache, 2-1, 2-5, 2-6, A-1
- commands
  - `#load`, 2-1, 2-2, A-1
  - `ahdl_include`, 2-1, A-1
- comment characters, 3-5
- compact models, 1-2
- compilation
  - process, 2-1
- compiled model libraries (CML), 2-1, 2-6, A-1, B-1
- compiler directives
  - `#include`, 2-9
- constant definitions, 3-2
- contribution operator, 3-2
- Datav Display, 1-7
- ddt operator, 3-1
- device, 1-3
  - definition name, 2-1
  - loading, 2-1
  - modifications, 1-8
  - parameters, 2-1
- directories
  - custom, 2-2
  - hpesof, 2-2
  - model file, A-1
  - site, 2-2
  - system include, 2-9
  - veriloga, 2-10
- disciplines, 3-1
- dynamic linking, 2-10
- environment variables
  - AGILENT_CML_BUILDER, B-1
  - AGILENT_MODEL_CACHE, 2-5
  - USER_VERILOGA_MODEL_PATH, 2-9
- error messages, 2-2, 2-7, 2-8, 2-10, 3-7
- event-driven constructs, 2-7
- examples, 3-1
  - capacitors and inductors, 3-3
  - example library, 1-2, 1-3
  - GaAs FET model, 1-4
  - nonlinear diode, 3-4
  - Parker-Skellern model, 1-8
  - resistor, 3-1
  - tutorial_PSFETV, 1-4
- explicit loading, 2-1, 2-2
- exponential function, 2-1
- file extensions

---

Index-1
.va, 2-2
.vams, 2-2
files
cds.lib, 1-2
constants.vams, 3-2, 3-8
disciplines.vams, 3-1
hpeesofsim.cfg, 2-6, 2-9
make specifications, A-1
vamake.spec, 2-6, A-1
flicker noise, 3-6
functions
$abstime, 2-7, 3-7
$bound_step, 3-8
$temperature, 3-2
$v_t, 3-5
exponential, 3-6
G
gcc compiler, B-1
H
Harmonic Balance, 2-7
hierarchy, 2-8
hpeesofsim.cfg file, 2-6, 2-9
HP-UX, B-1
I
idtmod operator, 2-7
improving compilation and load times, 2-6
intellectual property, A-1
internal node, 3-5
K
Kernighan & Ritchie compiler, B-1
keywords
analog, 3-2
begin and end, 3-5
ksh shell, B-1
L
libraries
compactLib, 1-2
Library Manager, 1-2
library path, 1-2
licenses
$sim_veriloga, 1-10
loading mechanisms, 2-1
M
macros, 3-2
mathematical operators, 3-5
memory states, 2-7
model
behavior, 3-1
cards, 2-1, 2-5
file directory, A-1
search path, 2-1
Model Writer, 1-2
module
block, 3-1
names, 2-8
parameters, 2-8
resolution, 2-8
multi-dimensional arrays, 2-8
N	namespaces, 2-2
natures, 3-1
netlist, 2-1, 2-2, 2-3, 2-9, 3-1
syntax errors, 2-3
nets, 3-5
network hierarchy, 2-3
nodes, 3-5
noise, 3-1, 3-2, 3-6
nonlinear behavior, 3-4
O
operators
contribution operator, 3-2
ddt, 3-1
idt, 3-8
idtmod, 2-7
out of range, 2-8
overriding built-in devices, 2-3
P
parameters, 2-8
parameter declaration, 3-7
parameter range, 3-7
phase-locked loop, 3-8
ports, 3-1
preprocessor
commands, 3-2
directives, 2-2
process design kits (PDK), 2-2, A-1
R
range specification, 2-8

S
search path, 2-1, 2-2, 2-3, 2-9, 2-10, A-1
series resistance, 3-5
shared libraries, 2-1, B-1
shot noise equation, 3-6
signal resolution, 3-8
sim_veriloga license, 1-10
simulator
  algorithms, 3-6
  execution time, 3-8
  simulation accuracy, 3-8
  simulation algorithms, 2-7
  simulation environment, 2-1
simulator configuration file, 2-9
slew filter, 2-7
special characters, 2-8
Spectre, 1-3, A-1
status messages, 2-2
sweep ranges, 3-8
swept parameters, 2-5, 3-7

T
temperature, 3-2, 3-5
test bench circuits, 3-8
thermal noise, 3-2, 3-6
thermal voltage, 3-5
time
  $abstime, 2-7, 3-7
  time-dependent functionality, 3-1
  transition filter, 2-7

U
unmatched parameters, 2-3
unsupported type message, 2-8
user-compiled models (UCM), 2-10

V
vamake.spec file, A-1
Verilog-A
  and UCM devices, 2-10
  debugger, 1-2
  fragments, 2-1
  make specifications file, 2-6
  system include, 2-9
  Tutorial, 1-3

W
warning messages, 2-7, 2-10
white noise, 3-2

Z
Z-transform filters, 2-7