



Agilent Technologies

Using Verilog-A in Advanced Design System

August 2005

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, Inc. 1983-2005
395 Page Mill Road, Palo Alto, CA 94304 U.S.A.

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.

SystemC[®] is a registered trademark of Open SystemC Initiative, Inc. in the United States and other countries and is used with permission.

Contents

1 Getting Started	
Using a Verilog-A Device in a Simulation	1-1
Modifying a Verilog-A Device.....	1-5
Overriding Model Parameters on Instances	1-7
Installing Verilog-A Based Devices Provided in a Design Kit.....	1-8
Licensing	1-12
2 Using Verilog-A with the ADS Analog RF Simulator (ADSSim)	
Loading Verilog-A modules.....	2-1
The Auto Loading Mechanism	2-1
Explicit Loading of Verilog-A modules	2-2
Overriding Built-in Devices	2-3
Using Models with Verilog-A Devices	2-4
The Verilog-A Compiled Model Library Cache	2-5
Controlling the Auto Compilation Process	2-7
Verilog-A Operator Limitations in Harmonic Balance and Circuit Envelope	2-7
Module and Parameter Naming.....	2-8
Parameters	2-8
Hierarchy and Module Resolution.....	2-9
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
4 Migrating from the SDD and UCM	
Symbolically Defined Devices	4-1
User-Compiled Models	4-8

A Verilog-A in ADS Design Kits

B Compilation Tools

Compiling Large Files on HP-UX..... B-1

Index

Chapter 1: Getting Started

Verilog-A devices provide all of the capabilities as well as the *look and feel* of traditional, built-in components, with the added benefit that the end-user can choose to modify the underlying equations. A number of new devices and models are supplied with Advanced Design System to provide both new model capability as well as to provide Verilog-A versions of models that already exist as built-in models.

This chapter provides an overview of the steps necessary to use Verilog-A devices. Many Verilog-A devices are provided as examples via a Design kit. For information on installing and using the Verilog-A devices supplied in the Verilog-A Design Kit, refer to [“Installing Verilog-A Based Devices Provided in a Design Kit” on page 1-8](#).

Using a Verilog-A Device in a Simulation

To illustrate how Verilog-A components are used and can be modified, a popular GaAs FET model that is also supplied in the Verilog-A Design Kit (see [“Installing Verilog-A Based Devices Provided in a Design Kit” on page 1-8](#)) is used in a tutorial project to show how a model can be simulated and modified. You can copy the Verilog-A Tutorial example project to your home directory or another preferred location. From the ADS Main window:

1. Choose **File > Copy Project**.
2. Using the Copy Project browser, click the **Example Directory** button and then the **Browse** button to quickly locate `Examples/Verilog-A/Tutorial_prj` as the source.
3. Specify your destination directory in the *To Project* field of the Copy Project browser and accept the default settings to *Copy Project Hierarchy* and *Open Project After Copy*.

After the project opens, a ReadMe.dsn schematic window will appear as shown in [Figure 1-1](#) below.

Verilog-A/Tutorial_prj	
This project contains designs used in the documentation. Open the design file or PUSH into the associated component.	
PSFETV example tutorial_PSFETV X1	Shows how to run and modify the Parker Skeltem FET.
SINEV example tutorial_SINEV X2	Demonstrates a Verilog-A source.
Overriding builtin example tutorial_resistor X6	Shows how to override a builtin component with a Verilog-A module.
SDD to Verilog-A GumPoon example tutorial_SDD_GumPoon X3	Compares the SDD and Verilog-A versions of GumPoon.
PLL system tutorial_PLL X5	A Verilog-A behavioral implementation of a PLL system.
Deadband amp tutorial_deadband X7	A Verilog-A example of a deadband amplifier.

Figure 1-1. The Verilog-A Tutorial_prj ReadMe.dsn

- The tutorial contains six different examples to illustrate Verilog-A components. Click the *PSFETV example* button on the left and then push down into the design.



- The example consists of a simple DC_FET curve tracer to sweep the device.

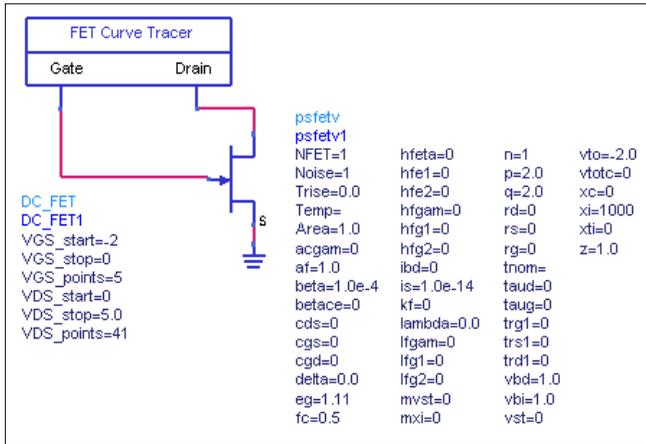


Figure 1-2. Simple Schematic Design using a Verilog-A Component

6. Run a simulation.

Note that the first time the project is simulated, the compiler will compile any un-compiled Verilog-A files found in the project's veriloga directory.

Note The compiled files, by default, reside in a cache directory in your \$HOME/hpeesof directory.

Status / Summary

```
AGILENT-VACOMP (*) 2003C.day Dec  3 2003 (built: 12/04/03 01:42:48)
Tiburon Design Automation (R) Verilog-A Compiler Version 0.97.120103.
Copyright (C) Tiburon Design Automation, Inc. 2002-2003. All rights reserved.

Compiling Verilog-A file
'/users/bobl/Tutorial_prj/veriloga/psfetv.va'
Loading Verilog-A module 'R' from
'/users/bobl/hpeesof/agilent-model-cache/cml/veriloga_21561_20031204_135236_007390/lib.hpux11/r
esv.cml'.

This module overrides the builtin 'R' (Linear Two Terminal Resistor).

HPEESOFSIM (*) 2003C.day Dec  3 2003 (built: 12/03/03 21:27:46)
Copyright Agilent Technologies, 1989-2003.
Loading Verilog-A module 'psfetv' from
'/users/bobl/hpeesof/agilent-model-cache/cml/veriloga_21561_20031204_135236_007390/lib.hpux11/p
sfetv.cml'.

CT DC_FET1.S1[1] <(GEMX netlist)>   DC_FET1.VGS=(-2->0)
DC DC_FET1.S1[1].DC_FET1.DC1[1/5] <(GEMX netlist)>   DC_FET1.VGS=-2   DC_FET1.VDS=(0->5)
..
.....
DC DC_FET1.S1[1].DC_FET1.DC1[2/5] <(GEMX netlist)>   DC_FET1.VGS=-1.5   DC_FET1.VDS=(0->5)
..
.....
DC DC_FET1.S1[1].DC_FET1.DC1[3/5] <(GEMX netlist)>   DC_FET1.VGS=-1   DC_FET1.VDS=(0->5)
..
.....
DC DC_FET1.S1[1].DC_FET1.DC1[4/5] <(GEMX netlist)>   DC_FET1.VGS=-500e-03   DC_FET1.VDS=(0->5)
..
.....
DC DC_FET1.S1[1].DC_FET1.DC1[5/5] <(GEMX netlist)>   DC_FET1.VGS=0   DC_FET1.VDS=(0->5)
..
.....
Resource usage:
    Total stopwatch time: 35.54 seconds.

-----
Simulation finished: dataset 'tutorial_PSFETV' written in:
    '/users/bobl/Tutorial_prj/data'
-----
```

If there had been a compile error, the status window would display the error and the associated line number. In this case, there were no errors and the simulation continued.

Note The status window provides information about the files that were compiled and the modules that were used. This is useful when files exist in multiple directories.

7. After the simulation is complete, open the Data Display *TestBench.dds* window if necessary. The display consists of a rectangular plot with a trace of the *IDS.i* data. Your results should look similar to [Figure 1-3](#).

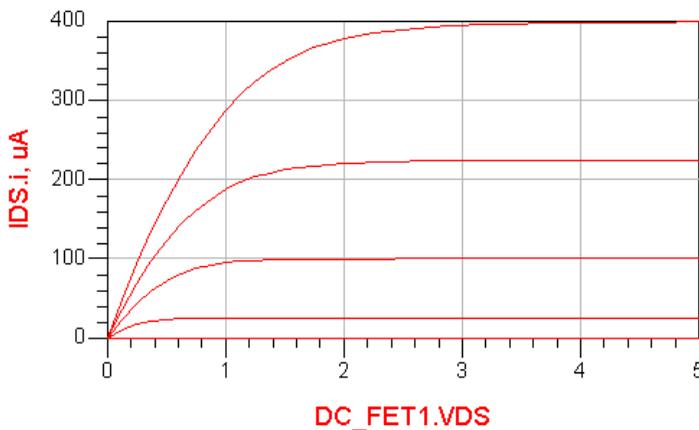


Figure 1-3. PSFET DCIV Results

The PSFETV (and any Verilog-A based device) can be used in all analyses that a traditional built-in device could be used. The PSFETV is defined in the file *psfetv.va* located in a directory called *veriloga* in the project directory. You can open and edit the file using any text editor. You can see from this file that the code is relatively easy to understand. The next section will show how to modify an equation and see the effect.

Other designs in the Tutorial project illustrate other key aspects of the Verilog-A language.

Modifying a Verilog-A Device

One powerful feature of Verilog-A is that a user can make modifications to the equations that describe the behavior of the device. These changes can be available in the simulator automatically, with no loss of analysis functionality. Many models will be distributed with their source code with the expectation 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, thereby improving simulation performance.

In this example, the PSFETV model will be modified slightly. During simulation, the program searches for the source code based on pre-defined search paths (discussed in detail later), in the project directory `veriloga`, or as specifically defined using a `VerilogA_Load` component.

1. If necessary, copy the project `Tutorial_prj` from the `Examples/Verilog-A` directory to a local directory. This tutorial contains a directory called `veriloga` that includes a file called `psfetv.va`. This file is a copy of the `parker_skellern.va` file distributed in the Verilog-A Design Kit, with the module name changed to `psfetv` to prevent unintended overwriting of the other model.
2. Open the design file `tutorial_PSFETV.dsn` and run a simulation to verify that the results are the same as the previous example.
3. Using any text editor, open the `psfetv.va` file.
4. Find the equation that describes the drain current,

```
Id: Id = Area * Beta_T * (1 + lambda * Vdst) * (pow(Vgt, q) - pow(Vgt - Vdt, q));
```

Purely for demonstration purposes, change the power law relation from “q” to “q/2” in the first power term (this has no physical meaning):

```
Id = Area * Beta_T * (1 + lambda * Vdst) * (pow(Vgt, q/2) - pow(Vgt - Vdt, q));
```

5. Save the file and start a simulation.

Note The program will detect that the file has not been compiled and will compile the source file. It will also compile other Verilog-A files found in the project’s `veriloga` directory if they have yet to be compiled, or if they are out of date.

The Data Display window now shows very different results (see [Figure 1-4](#)) compared to [Figure 1-3](#), demonstrating that the modification to the equation did indeed take effect.

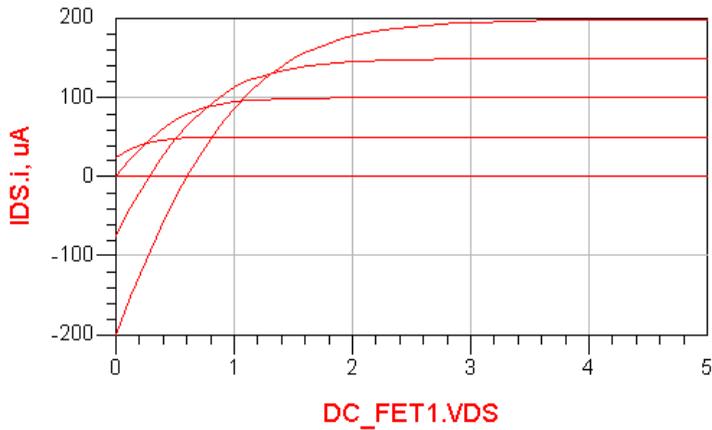


Figure 1-4. PSFETV_DCIV Results using Power Law Relation of $q/2$

Overriding Model Parameters on Instances

One feature that most Verilog-A models implemented for ADS have is that the instance component can share all of the model parameter definitions. In most cases, the instance parameters are not visible by default, but can be edited (and made visible) from the *Edit Component Parameters* dialog box. Instance parameters will override the associated model parameter values. This feature provides a method to easily evaluate device mismatch, for example, since only the model parameter that is being modified needs to be changed at the instance level.

Note Typically, some parameters are only available on the instance, and some parameters are only available on the model. For example, T_{nom} would only make sense on a model card, and $Trise$, the instance temperature rise over ambient, would only make sense on an instance. For this reason, the AEL definition hides these parameters from the schematic, although they are available at the netlist level.

The underlying Verilog-A definition of the device is stored in a text file and, if necessary, compiled by the simulator during a simulation according to a flexible and configurable set of rules, described later.

Installing Verilog-A Based Devices Provided in a Design Kit

Design Kits provide a convenient method for model developers to distribute new models, including those developed using Verilog-A. The use model for installing and using Verilog-A models distributed in Design Kits is the same as any other models distributed as ADS Design Kits. For more information, refer to the *Design Kit Installation and Setup* documentation.

After properly installing the Design Kit, the components are made available on the associated palettes. The ADS release includes a Design Kit called TIBURONDA_VERILOGA_DESIGN_KIT delivered as an unzipped file in the following location:

```
$HPEESOF_DIR/tiburon-da/ads/designkits/tiburon-da_veriloga
```

This design kit includes the nonlinear models listed in [Table 1-1](#) and [Table 1-2](#). Models shown in [Table 1-1](#) are new to, or later versions of, existing models in ADS.

Note All of these models are proof-of-concept models only and have not undergone the exhaustive qualification that built-ins do.

Table 1-1. Verilog-A Versions of Models not yet Available as Built-in Devices in ADS.

Model Name	Description
aTFT	Shur-RPI amorphous silicon thin film transistor MOSFET model
BSIMSOI	Version 3.1 of the BSIMSOI model family
EKV	Version 2.6 of the EKV MOSFET model
HICUM_L0	Level 0 version of the HICUM BJT model, a simplified version of the full HICUM model.
HiSIM	STARC/Hiroshima University surface potential based MOSFET model
Parker_Skellern	The Parker-Skellern MESFET model for medium power applications.
pTFT	Shur-RPI polysilicon thin film transistor MOSFET model

The models listed in [Table 1-2](#) are Verilog-A examples of models that already exist as built-in devices.

Table 1-2. Verilog-A Versions of Models Already Available in ADS.

Model Name	Description
Angelov	Angelov GaAs FET model
BJT	SPICE Gummel-Poon BJT model
BSIM3	UC Berkeley BSIM3 (version 3.22)
BSIM4	UC Berkeley BSIM4 (version 4.3)
Curtice	Curtice Quadratic GaAs FET model
Diode	SPICE diode
JFET	SPICE silicon junction FET
JunCap	Philips JunCap diode
MESFET	SPICE Metal Semiconductor FET model
MEXTRAM	Philips Most Exquisite Transistor model for BJT devices
MOS9	Philips MOS Model 9
MOS11	Philips MOS Model 11
TOM1	Triquint's Own Model 1
TOM3	Triquint's Own Model 3
VBIC	Vertical Bipolar InterCompany BJT model, as released.

Use the following procedure to install the TIBURONDA_VERILOGA_DESIGN_KIT. For more detailed information on installing and using ADS Design Kits, refer to the *Design Kit Installation and Setup* manual.

From the ADS Main window,

1. Choose **DesignKit > Install Design Kits** to display the *Install ADS Design Kit* dialog box.
2. Since the Design Kit is delivered as an unzipped file, simply click the Path **Browse** button under *2. Define Design Kit* in the Install ADS Design Kit dialog box. Use the *Select Design Kit Directory* dialog box to select the following Design Kit:

\$HPEESOF_DIR/tiburon-da/ads/designkits/tiburon-da_veriloga

3. Click **OK** in the *Select Design Kit Directory* dialog box. All associated Design Kit information (i.e. Path, Name, Boot File, and Version) should automatically populate the appropriate fields in the Install ADS Design Kit dialog box.
4. Change the Design Kit installation level from the default USER LEVEL (if desired) and click **OK**. The program will then install the Design Kit and return with a dialog box to indicate successful installation.
5. Close the Install Design Kit dialog box and open a schematic window. The *Devices-Verilog-A* palette should now be available in the Component Palette List. [Figure 1-5](#) shows the icons for each of the devices and models available in the *Devices-Verilog-A* palette.

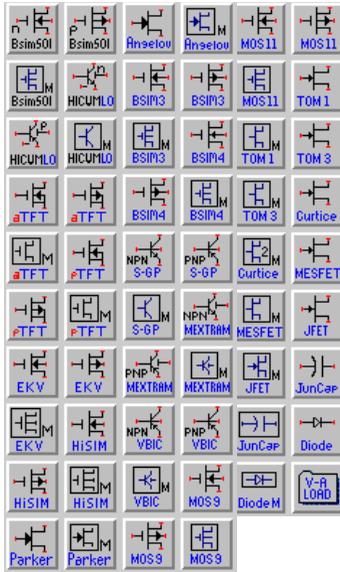


Figure 1-5. Devices-Verilog-A Component Palette

Note To prevent unintentional overriding of the built-in versions of these models, the Verilog-A modules name use an `_va` suffix for the name.

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:** On the compilation of the first Verilog-A file. 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 `#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 the ADS Analog RF Simulator (ADSSim)

Once Verilog-A modules are loaded into the simulation environment, they may be used just like any other device in the system. A module loaded into the system takes the look and feel of a built-in ADS Analog RF Simulator (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.

Loading Verilog-A modules

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 are explained below in detail.

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, you do not need to be concerned about the compilation process. The process has been designed to allow 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.

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,

```
PRJ_DIR/veriloga  
$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.

If you want the Verilog-A module to only have visibility within the current project, then create a Verilog-A directory under the current project (`PRJ_DIR/veriloga`) and copy the Verilog-A file to that location. Putting a Verilog-A module in the local user directory (`$HOME/hpeesof/veriloga`) will make it visible to simulations in all projects for this user account. The site custom directory (`$HPEESOF_DIR/custom/veriloga`) should be used to make Verilog-A modules available to all users of a particular ADS installation. Finally, the site directory (`$HPEESOF_DIR/veriloga`) should not be used by you as an end user, this directory is used by Agilent Technologies to deliver new Verilog-A modules as part of the ADS system. Note that Verilog-A files placed in the site directory will not be compiled.

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

Explicit loading refers to explicitly loading all modules in a named Verilog-A file using the load component. The load component is available with the Verilog-A Design Kit.

To load a file:

1. Insert a *VerilogA_Load* component from the *Devices-Verilog-A* palette.
2. Use the *Edit Parameters* dialog box to add the name of the file to the Module File Name parameter. Note that this parameter is repeatable to enable multiple files to be loaded. The component should look similar to [Figure 2-1](#).



Figure 2-1. The VerilogA_Load Component

When the file name specified is an absolute name, the system loads the file directly. When the file name is relative, the system searches each directory in the Verilog-A model search path and loads the first file found with that name. Note that all modules in a file are loaded (unlike auto loading where only the required module is loaded into the system.)

The load component netlists the `#load` ADS preprocessor command which has the syntax:

```
#load "veriloga", "filename"
```

`#load` is processed early in the simulation boot phase. `#load` commands may be placed anywhere in the netlist, but since they are preprocessor commands, there is no regard for network hierarchy. `#load` processing occurs before auto loading; 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, 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 simple linear resistor). When a built-in device is overridden, the system issues a status message warning of the override as shown below.

```
Loading Verilog-A module 'R' from  
'/users/bobl/hpeesof/agilent-model-cache/cml/veriloga_21561_20031204_13  
5236_007390/lib.hpux11/my_resistor.cml'.
```

```
This module overrides the builtin 'R' (Linear Two Terminal Resistor).
```

Note Overriding built-ins is a powerful and convenient way to use Verilog-A based devices in place of an existing device, but 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. 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 ADS 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 ADS must be run concurrently (perhaps one from the ADS 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 the following status message,

```
HPEESOFSIM (*) 2003C.day Dec 13 2003 (built: 12/13/03 21:28:34)
Copyright Agilent Technologies, 1989-2003.
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
'/users/bobl/QA_Test/Tutorial_prj/veriloga/deadband.va'
AGILENT-VACOMP (*) 2003C.day Dec 13 2003 (built: 12/14/03 01:50:07)
Tiburon Design Automation (R) Verilog-A Compiler Version 0.99.121203.
Copyright (C) Tiburon Design Automation, Inc. 2002-2003. All rights
reserved.

Compiling Verilog-A file
'/users/bobl/QA_Test/Tutorial_prj/veriloga/gumpoov.v'
AGILENT-VACOMP (*) 2003C.day Dec 13 2003 (built: 12/14/03 01:50:07)
Tiburon Design Automation (R) Verilog-A Compiler Version 0.99.121203.
Copyright (C) Tiburon Design Automation, Inc. 2002-2003. All rights
reserved.
```

CML files vary in size depending on the size of the Verilog-A file and the platform used. It is recommend that you locate the cache on a local disk with at least 200Mbytes 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

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*.) 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`, the cache is not used and CML files are written to the platform specific directories:

```
lib.hpux11  
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 zero (0) in Harmonic Balance analysis. 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

ADS 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 ADS 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 ADS 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; ADS arrays can be multi-dimensional. If a multidimensional 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 enables you to describe components in terms of their structure or their behavior or a mixture of both. Often, Verilog-A modules in a module hierarchy are all contained in the same Verilog-A file. However, this is not a requirement. The fact that the modules are in the same file has no special importance. That is, in terms of what modules actually get instantiated when a hierarchical module is itself instantiated in the ADS netlist is set by the search paths and not by what modules are contained in what file.

When a Verilog-A module is instantiated, the system works recursively 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, `hpeesofsim.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:

```
$HPPEESOF_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 `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.

```
[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
```

Line [1] instructs the compiler to insert the contents of the file *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 *R* and has two ports, named “p” and “n”. Ports provide connections to other modules.

Line [3] declares that the ports *p* and *n* have the *nature* of those declared in the *electrical* discipline, as defined in the *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 *R*, and assigns it a default value of 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 *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 cannot 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 $4 * K * T / R$. 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.

```

[1] `include "disciplines.vams"
[2] `include "constants.vams"
[3] module R(p,n);
[4]   electrical p,n;
[5]   parameter real R=50.0;
[6]   analog V(p,n) <+ R * I(p,n) + white_noise(4 * `P_K * $temperature / R,
"thermal");
[7] endmodule

```

Note that line [6] of the example code above shows the added noise.

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,

$$I = C * dV / dt$$

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 `ddt()` operator. The model then becomes,

```

`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

```

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,

$$V = L * dI/dt$$

and the source code is:

```

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

```

```

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

```

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,

$$I = I_s * (\exp(V/V_{th} - R_s * I) - 1)$$

The implementation is shown below.

```

`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 * (exp(Vd / (N * $vt - Rs * I(anode, cathode)) / $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, cathode) <+ Id + ddt(Qd);
    end
endmodule

```

```
end
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:

```
`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));
```

```

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.

```

// 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.

```

// 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,

```
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,

```
except 5
```

will not allow the value of 5 to be passed.

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

```
parameter real X = 20.0 from (-inf: -10] from [10:inf);
```

can also be written as,

```
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:

```
`include "disciplines.vams"  
`include "constants.vams"  
module sine_wave(n1,n2);  
    electrical n1,n2;
```

```
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:

```
// 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
```

```

// 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). ADS will not.

The `phaseDetector` makes use of an analog function definition of `chopper` to simplify the code. Analog functions can be thought of a 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;

```

```

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.

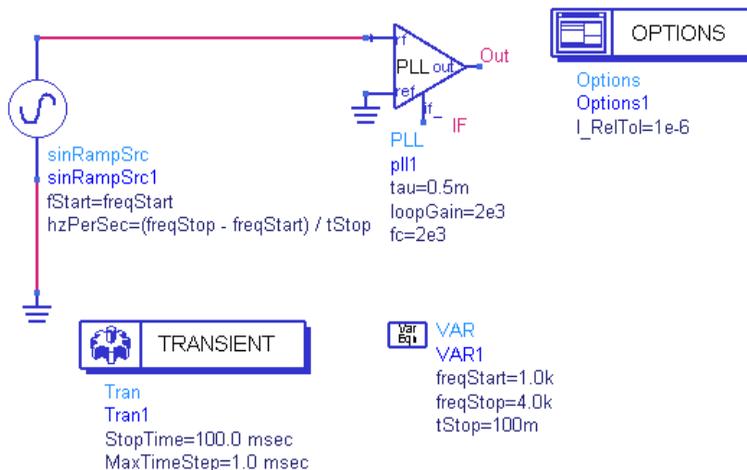


Figure 3-1. Phased-Locked Loop Test Circuit

Plotting the Out node shows the VCO control voltage response to the swept frequency input, indicating the locking range of the circuit.

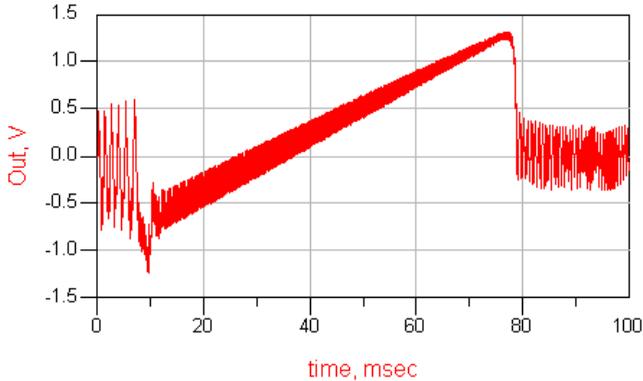


Figure 3-2. VCO Locking Range

Using Hierarchy to Manage Model Complexity

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);
    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.

Chapter 4: Migrating from the SDD and UCM

Symbolically defined devices (SDDs) and the user-compiled model (UCM) interface provide alternate methods to implement and distribute models with their own unique advantages. It is possible for advanced users to translate these models into Verilog-A with a fraction of the original effort required to develop the models. Unfortunately, due to the flexible natures of these interfaces, it is not possible to provide an automatic translation capability. However, this chapter will provide a general discussion of the issues and steps necessary to translate a model. The chapter will only deal with generating the Verilog-A source code, as the other aspects of the model translation (AEL, symbols, etc.) are the same as with other models.

Symbolically Defined Devices

The SDD interface is similar to Verilog-A. The SDD describes the model in terms of charges and currents and automatically generates the derivatives and load information, as does Verilog-A. However, SDDs automatically determine the relationship between equations whereas Verilog-A is a procedural language where the order of the equations is important. As an example, the Gummel_Poon SDD model provided in the `Examples/Tutorial/SDD_Examples_prj` will be used to illustrate the necessary steps. To begin,

1. Copy the *SDD_Examples_prj* to a local directory.
2. Open a new schematic window and insert the *GumPoon* subcircuit.
3. Start a simulation. The simulator will report that no simulation component has been specified and will stop. This can be ignored as the step was used to create the netlist.
4. Copy the *netlist.log* file found in the project directory to a new file named *gumpoon.va* and open the file.
5. The first line of the *gumpoon.va* file can be deleted. Add the necessary include files and module definition block as shown below. Note that three internal nodes, *bi*, *ci*, and *ei*, have been defined.

```
`include "disciplines.vams"  
`include "constants.vams"  
module gumpoon(b, c, e);  
    electrical b, c, e, bi, ci, ei;  
    analog begin
```

```
// SDD netlist fragment
end
endmodule
```

The subcircuit parameter list provides a convenient starting point to convert parameters to Verilog-A using word processor functions to substitute a comma for the spaces between parameters:

```
parameter real n=1, xti=3, Rth=0, Tau=0, Tamb=27, eg0=1.16, xtb=1, rbm=1e-12,
jrb=10e-3, mje=0.5, mjc=0.333, vjc=585.8e-3, vtf=1.124, vje=1.993, nf=1.002,
nr=0.9649, ne=1.630, nc=1.055, tf=14.19e-12, tr=1e-9, cje=1.178e-12,
cjc=392.9e-15, jtf=914.3e-3, xtf=32.13, rb=6.549, rc=6.314, re=.15,
vbf=32.37, vbr=10, jbr=8.26e-3, jbf=914.3e-3, jle=0, jlc=0, bf=127, br=10,
js=16.55e-15;
```

The `_vn` quantities for the SDD correspond to system values of the port values. These are converted to voltage accesses and should be moved to the beginning of the module:

```
analog begin
  vei=V(ei);
  vci=V(ci);
  vbi=V(bi);
  ve=V(e);
  vc=V(c);
  vb=V(b);
```

Note that the SDD is defined with the port pin grounded so the Verilog-A voltages are implicitly referenced to ground.

The SDD:SDD1 line corresponds to the load contribution of a Verilog-A device. This line can be moved to the end of the file and converted to contribution statements. The statements of the form `I[...]` are explicit current port relations and easily convert to contribution statements. For example,

```
I[1,0]=ib+_v1*1e-12
```

becomes

```
I(b) <+ ib + V(b)*1e-12;
```

and contributions with weighting functions of 1 correspond to applying the `d/dt` operator, so

```
I[1,1]=qb2+qbe
```

becomes

```
I(b) <+ ddt(qb2+qbe);
```

The implicit equations of the form $F[\dots]$ relate currents and voltages at ports with an implicit form $F[\dots]=0$. So

```
F[4,0]=vbi+ib*Rbb-vb
F[4,1]=(qb2+qbe)*Rbb
```

is rearranged as

```
vbi-vb= ib*Rbb-ddt(qb2 + qbe)*Rbb
```

and is implemented as

```
V(bi,b) <+ Rbb*(ib + ddt(qb2 + qbe)) ;
```

This process is repeated for the other contributions.

The SDD uses a function named *diode* to implement the diode junction relation. This function, and the exponential function *exp_soft*, can easily be converted to analog functions. For example,

```
diode(v,is,n)=is*(exp_soft(v/(n*vt)) - 1)
```

becomes

```
analog function real diode;
  input v, is, n, vt;
  real v, is, n, vt;
  begin
    diode = is * (exp_soft(v / (n * vt)) - 1);
  end
endfunction
```

while

```
exp_soft(x)=if (x < exp_max) then exp(x) else (x+1-exp_max)*exp(exp_max)
endif
```

becomes

```
analog function real exp_soft;
  input x;
  real x;
  `define max_arg ln(1e16)
  begin
    if (x < `max_arg)
      exp_soft = exp(x);
    else
      exp_soft = (x + 1 - `max_arg) * exp(`max_arg);
    end
  end
endfunction
```

Note that Verilog-A requires that the analog function *exp_soft* be defined prior to its use in the analog function *diode*. Also, the variable *max_arg* is converted to a macro definition.

At this point, the source code is as shown below:

```

`include "disciplines.vams"
`include "constants.vams"
`define max_arg ln(1e16)

module gumpoon(b, c, e);
    electrical b, c, e, bi, ci, ei;
    parameter real n=1, xti=3, Rth=0, Tau=0, Tamb=27, eg0=1.16, xtb=1,
    rbm=1e-12, jrb=10e-3, mje=0.5, mjc=0.333, vjc=585.8e-3, vtf=1.124,
    vje=1.993, nf=1.002, nr=0.9649, ne=1.630, nc=1.055, tf=14.19e-12,
    tr=1e-9, cje=1.178e-12, cjc=392.9e-15, jtf=914.3e-3, xtf=32.13,
    rb=6.549, rc=6.314, re=.15, vbf=32.37, vbr=10, jbr=8.26e-3,
    jbf=914.3e-3, jle=0, jlc=0, bf=127, br=10, js=16.55e-15;
    analog function real exp_soft;
        input x;
        real x;
        begin
            if (x < `max_arg)
                exp_soft = exp(x);
            else
                exp_soft = (x + 1 - `max_arg) * exp(`max_arg);
            end
        endfunction
    analog function real diode;
        input v, is, n, vt;
        real v, is, n, vt;
        begin
            diode = is * (exp_soft(v / (n * vt)) - 1);
        end
    endfunction

    real vei, vci, vbi, ve, vc, vb;

    analog begin
        vei=V(ei);
        vci=V(ci);
        vbi=V(bi);
        ve=V(e);
        vc=V(c);
        vb=V(b);

        vjc_T=vjc
        vje_T=vje
        Br_T=br
        Eg(t)=eg0-7.02e-4*(t+TzeroC)^2/(1108+(t+TzeroC))
    end
endmodule

```

```

Bf_T=bf
Js_T=js*TempRatio^xti *
      exp_soft(Eg(Tamb)*((Tj-Tamb)/((Tamb+TzeroC)*vt)))
TempRatio=(Tj+TzeroC)/(Tamb+TzeroC)
TzeroC=273.15
Tj=Tamb
ibb=if (ib < lpA ) then lpA else ib endif
Rbb=3*(rb - rbm)*(tan(Z) - Z)/(Z*tan(Z)^2) + rbm
Z=(-1 + sqrt(144*ibb/(pi^2*jrb) + 1))/((24/pi^2)*sqrt(ibb/jrb))
qbc=tr*diode(vbci, Js_T, nr) -
cjc*vjc_T*((1-vbci/vjc_T)^(1-mjc))/(1-mjc)
qb2=qbc
iff=diode(vbei, Js_T, nf)
TFF=tf*(1+xtf*(exp(vbci/(1.44*vtf))*(iff/(iff+jtf))^2))
qbe=diode(vbei, TFF*js/QB, nf) -
      cje*vje_T*((1-vbei/vje_T)^(1-mje))/(1-mje)
ib2=diode(vbei, jle, ne) + diode(vbci, jlc, nc)
ib1=diode(vbei, Js_T/Bf_T, nf) + diode(vbci, Js_T/Br_T, nr)
ic2=-diode(vbci, Js_T/Br_T, nr) - diode(vbci, jlc, nc)
ic1=diode(vbei, Js_T/QB, nf) - diode(vbci, js/QB, nr)
QB=(Q1/2)*(1 + sqrt(1+4*Q2))
vt=boltzmann*(Tamb+TzeroC)/qelectron
Q2=diode(vbei, Js_T/jbf, nf) + diode(vbci, Js_T/jbr, nr)
Q1=1/(1 - vbci/vbf - vbei/vbr)
vbei=vbi-vei
vbe=vb-ve
vbc=vb-vc
vbci=vbi-vci
ib=ib1+ib2
ic=ic1+ic2
ie=-(ib+ic)
X=1.0 Hz
endmodule

```

The remaining equations can easily be converted to valid Verilog-A equations by appending semi-colons and declaring the variables. Certain equations, such as the equations for *Tamp*, *vt*, and *TzeroC* can be updated to system parameter function calls or macro constants. Other equations, such as *X=1.0Hz*, are superfluous and can be deleted. The final step is to rearrange the order of the equations. The compiler will issue a message indicating a *memory state* exists. This is a valid outcome but is also indicative of an un-initialized variable, which, in turn, indicates that the equation declaring the variable should be moved before that equation. The final code is shown below.

```

`include "disciplines.vams"
`include "constants.vams"
`define max_arg ln(1e16)

```

```

`define TzeroC `P_CELSIUS0
module gumpoon(b, c, e);
    electrical b, c, e, bi, ci, ei;
    parameter real n=1, xti=3, Rth=0, Tau=0, Tamb=27, eg0=1.16, xtb=1,
    rbm=1e-12, jrb=10e-3, mje=0.5, mjc=0.333, vjc=585.8e-3, vtf=1.124,
    vje=1.993, nf=1.002, nr=0.9649, ne=1.630, nc=1.055, tf=14.19e-12, tr=1e-9,
    cje=1.178e-12, cjc=392.9e-15, jtf=914.3e-3, xtf=32.13, rb=6.549, rc=6.314,
    re=.15, vbf=32.37, vbr=10, jbr=8.26e-3, jbf=914.3e-3, jle=0, jlc=0,
    bf=127, br=10, js=16.55e-15;
    analog function real exp_soft;
    input x;
    real x;
    begin
    if (x < `max_arg)
        exp_soft = exp(x);
    else
        exp_soft = (x + 1 - `max_arg) * exp(`max_arg);
    end
endfunction
analog function real diode;
input v, is, n, vt;
real v, is, n, vt;
begin
    diode = is * (exp_soft(v / (n * vt)) - 1);
end
endfunction

analog function real Eg;
input t, eg0;
real t, eg0;
begin
    Eg=eg0-7.02e-4*(t+`TzeroC)*(t+`TzeroC)/(1108+(t+`TzeroC));
end
endfunction

real vei, vbe, vbei, vbci, vbc, vci, vbi, ve, vc, vb;
real Tj, vjc_T, vje_T, Br_T, Bf_T, Js_T, TempRatio, vth;
real ic, ie, ib, ibb, Z, vt, pi2, Rbb, Q1, Q2;
real qbc, qb2, iff, TFF, qbe, ib2, ib1, ic1, ic2, QB;

analog begin
    vei=V(ei);
    vci=V(ci);
    vbi=V(bi);
    ve=V(e);
    vc=V(c);
    vb=V(b);
    vbei=vbi-vei;
    vbe=vb-ve;
    vbc=vb-vc;

```

```

vbci=vbi-vci;
Tj=$temperature;
vth = $vt;
vjc_T=vjc;
vje_T=vje;
Br_T=br;
Bf_T=bf;
TempRatio=(Tj+'TzeroC)/(Tamb+'TzeroC);
Js_T=js*pow(TempRatio,xti) *
    exp_soft(Eg(Tamb,eg0)*((Tj-Tamb)/((Tamb+'TzeroC)*vth)));
qbc=tr*diode(vbci, Js_T, nr, vth) -
    cjc*vjc_T*(pow(1-vbci/vjc_T,1-mjc))/(1-mjc);
qb2=qbc;
iff=diode(vbei, Js_T, nf, vth);
TFF=tf*(1+xtf*(exp(vbci/(1.44*vtf)*pow(iff/(iff+jtf),2))));
ib2=diode(vbei, jle, ne, vth) + diode(vbci, jlc, nc, vth);
ib1=diode(vbei, Js_T/Bf_T, vth) + diode(vbci, Js_T/Br_T, nr, vth);
ic2=-diode(vbci, Js_T/Br_T, nr, vth) - diode(vbci, jlc, nc, vth);
Q1=1/(1 - vbci/vbf - vbei/vbr);
Q2=diode(vbei, Js_T/jbf, nf, vth) + diode(vbci, Js_T/jbr, nr, vth);
QB=(Q1/2)*(1 + sqrt(1+4*Q2));
qbe=diode(vbei, TFF*js/QB, nf, vth) -
    cje*vje_T*(pow(1-vbei/vje_T,1-mje))/(1-mje);
ic1=diode(vbei, Js_T/QB, nf, vth) - diode(vbci, js/QB, nr, vth);
ib=ib1+ib2;
if (ib < 1e-12 )
    ibb = 1e-12;
else
    ibb = ib;
pi2 = 'M_PI * 'M_PI;
Z=(-1 + sqrt(144*ibb/(pi2*jrb) + 1))/((24/pi2)*sqrt(ibb/jrb));
Rbb=3*(rb - rbm)*(tan(Z) - Z)/(Z*pow(tan(Z),2) + rbm;
ic=ic1+ic2;
ie=-(ib+ic);
V(bi,b) <+ Rbb * (ib + ddt(qb2+qbe));
V(e,ei) <+ re * (ie - ddt(qbe));
V(c,ci) <+ rc * (ic - ddt(qb2));
I(c) <+ -ddt(qb2) + ic+vc*1e-12;
I(b) <+ ddt(qb2+qbe) + ib+vb*1e-12;
I(e) <+ -ddt(qbe) + ie+ve*1e-12;
end
endmodule

```

User-Compiled Models

Translating user-compiled models (UCMs) is more complicated both due to the additional information in UCM source files, as well as the additional freedom (and, therefore, variations) that C-code provides. Because the UCM interface requires separate loads for transient simulations, this usually forces the model author to modularize the code, making it easier to find the code that needs to be translated.

The PNDIODE UCM distributed in the `examples/Tutorial/UserCompiledModel_prj` project will be used as an example. `PNDIODE.c` is the main file. The functions to consider are the `analyze_lin`, `analyze_nl`, `analyze_ac_n`, and `diode_nl_iq_gc` functions. The `analyze_lin` function calculates and loads the linear parts of the nonlinear diode model while the `analyze_nl` function calls `diode_nl_iq_gc` to calculate the values to load. These functions will be easy to map to corresponding contribution statements. The `analyze_ac_n` function includes the noise contributions and can also easily be mapped.

The `diode_nl_iq_gc` function is typical of UCM code structures in that most of the model evaluation is done in this one function. Stripping the function declaration information, this function becomes:

```
csat = IS_P * AREA_P;
vte = N_P * VT;
vd = vPin[2] - vPin[1]; /* junction voltage */
/*
 * compute current and derivatives with respect to voltage
 */
if ( vd >= -5.0*vte )
{
    exparg = ( vd/vte < 40.0 ) ? vd/vte : 40.0;
    evd = exp(exparg);
    *id = csat * (evd - 1.0) + GMIN * vd;
    *gd = csat * evd / vte + GMIN;
}
else
{
    *id = -csat + GMIN * vd;
    *gd = -csat / vd + GMIN;
    if ( BV_P != 0.0 && vd <= (-BV_P+50.0*VT) )
    {
        exparg = ( -(BV_P+vd)/VT < 40.0 ) ? -(BV_P+vd)/VT : 40.0;
        evrev = exp(exparg);
        *id -= csat * evrev;
        *gd += csat * evrev / VT;
    }
}
```

```

}
/*
 * charge storage elements
 */
fcpb = FC_P * VJ_P;
czero = CJO_P * AREA_P;
if (vd < fcpb)
{
  arg = 1.0 - vd / VJ_P;
  sarg = exp(-M_P * log(arg));
  *qd = TT_P * (*id) + VJ_P * czero * (1.0 - arg * sarg) / (1.0 - M_P);
  *capd = TT_P * (*gd) + czero * sarg;
}
else
{
  xfc = log(1.0 - FC_P);
  f1 = VJ_P * (1.0-exp((1.0-M_P)*xfc)) / (1.0-M_P);
  f2 = exp((1.0+M_P)*xfc);
  f3 = 1.0 - FC_P * (1.0+M_P);
  czof2 = czero / f2;
  *qd = TT_P * (*id) + czero * f1 + czof2 * (f3 * (vd - fcpb) + (M_P /
(VJ_P + VJ_P)) * (vd * vd - fcpb * fcpb));
  *capd = TT_P * (*gd) + czof2 * (f3 + M_P * vd / VJ_P);
}

```

The first step is to remove any capacitance and conductance equations that were used to load the off diagonal elements in the Jacobian. Next, convert `{ / }` blocks to *begin / end* blocks and convert any math functions and operators to their Verilog-A equivalent.

Note In Verilog-A *log* is *log base 10* whereas in the C language it is *natural log*.

The diode terminals will be called anode, cathode, and internal. Access to the *vPin* array will be converted to voltage access functions and the parameter macros will have their *_P* suffix stripped.

The parameter declarations can be generated by using the `PNDIODE.ael create_item`'s declaration. Temperature dependent variables, such as *VT*, can be mapped to `$vt`, `GMIN` to a macro definition, and the double declaration of variables in the C-code can be mapped to real declarations in Verilog-A. The final code then becomes:

```

`include "constants.vams"
`include "disciplines.vams"
`define GMIN 1.0e-12
module PNDIODE(anode,cathode);

```

```

inout 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 EG = 0 from (0:inf]; //Activation energy [eV]
parameter real XTI = 3.0 from [0:inf]; //IS temp exp.
parameter real KF = 0.0; //Flicker noise coef
parameter real AF = 0.0 from (0:inf]; //Flicker noise exponent
parameter real FC = 0.0 from [0:1]; //Forward bias junct parm
parameter real BV = 0.0 from [0:inf]; //Reverse breakdown voltage [v]
parameter real IBV = 0.0 from [0:inf]; //Current at BV [A]
real vd, csat, vte, evd, evrev;
real id, qd, exparg, vth, T;
real fcpb, xfc, f1, f2, f3;
real czero, arg, sarg, czof2;
analog begin
  csat = IS * AREA;
  vth = $vt;
  vte = N * vth;
  vd = V(anode, internal); /* junction voltage */

  /*
  * * compute current and derivatives with respect to voltage
  * */
  if ( vd >= -5.0*vte ) begin
    exparg = ( vd/vte < 40.0 ) ? vd/vte : 40.0;
    evd = exp(exparg);
    id = csat * (evd - 1.0) + `GMIN * vd;
  end
  else begin
    id = -csat + `GMIN * vd;
    if ( BV != 0.0 && vd <= (-BV+50.0*vth) ) begin
      exparg = ( -(BV+vd)/vth < 40.0 ) ? -(BV+vd)/vth : 40.0;
      evrev = exp(exparg);
      id = id - csat * evrev;
    end
  end
end
/*
* * charge storage elements
* */
fcpb = FC * VJ;
czero = CJO * AREA;

```

```

if (vd < fcpb) begin
  arg = 1.0 - vd / VJ;
  sarg = exp(-M * ln(arg));
  qd = TT * id + VJ * czero * (1.0 - arg * sarg) / (1.0 - M);
end
else begin
  xfc = ln(1.0 - FC);
  f1 = VJ * (1.0-exp((1.0-M)*xfc)) / (1.0-M);
  f2 = exp((1.0+M)*xfc);
  f3 = 1.0 - FC * (1.0+M);
  czof2 = czero / f2;
  qd = TT * id + czero * f1 + czof2 * (f3 * (vd - fcpb) + (M / (VJ + VJ))
* (vd * vd - fcpb * fcpb));
end
I(anode, internal) <+ id;
V(internal, cathode) <+ I(internal, cathode) * (RS / AREA);
I(anode, internal) <+ ddt(qd);
// Noise
T = $temperature;
V(internal, cathode) <+ white_noise(4 * `P_K * T * (RS / AREA),
"thermal");
I(anode, internal) <+ white_noise(2 * `P_Q * id, "thermal");
if (id > 0)
  I(anode, internal) <+ flicker_noise(KF * pow(id, AF), 1.0, "flicker");
else
  I(anode, internal) <+ flicker_noise(KF * -pow(-id, AF), 1.0,
"flicker");
end
endmodule

```

Although it is not possible to predict what code style a UCM author might use, it is possible to convert a model to Verilog-A.

Appendix A: Verilog-A in ADS Design Kits

This appendix describes the steps necessary to distribute Verilog-A models in an ADS Design Kit. Design Kits provide a convenient way to distribute Verilog-A models and the steps are very similar to the steps to create conventional Design Kits.

For an overview of the process and the specific steps involved in creating an ADS Design Kit, refer to the *Design Kit Development* manual.

Verilog-A models require a similar infrastructure as User-Compiled Models (UCM). Because the Verilog-A interface includes the appropriate tools to recompile on the end user's platform, Verilog-A models can be distributed as source code, compiled form (Compiled Model Library, or CML), or both. As with the UCM, the compiled object library must be provided for each platform supported by Advanced Design System.

Verilog-A modules in a Design Kit are automatically made available to projects using that design kit. In this release, the Verilog-A files must be placed in the platform specific design kit User-Compiled Model directory:

```
design_kit/bin/$sarch
```

where `$sarch` represents the platform architecture (win32, hpux11, sun57, or linux_x86). If you want to distribute the Verilog-A source, then that is all that needs to be done. If you want to ship CML files then further steps are required. You may want to ship only CML if you want to protect your intellectual property. Alternatively, you may want to ship both Verilog-A source and CML files. This will avoid having all Design Kit users go through the compilation step the first time they simulate with your kit, and it will provide the ability to view and edit Verilog-A source code.

Lets assume you have a number of Verilog-A files in the Design Kit as,

```
dk_prj/bin/$sarch/resistor.va
```

```
dk_prj/bin/$sarch/varactor.va
```

By default, the CML files associated with these Verilog-A sources are stored in the CML cache (see [“The Verilog-A Compiled Model Library Cache” on page 2-5](#)). You can have the CML files written locally by adding a `vamake.spec` file to the Verilog-A source directory. Add a single line to `vamake.spec` as,

```
USE_CACHE=NO
```

From now on, all CML files will be written locally and assuming you have simulated on HP-UX, your directory structure will look like this,

```

dk_prj/bin/hpux11/vamake.spec
dk_prj/bin/hpux11/resistor.va
dk_prj/bin/hpux11/varactor.va
dk_prj/bin/hpux11/lib.hpux11/cml-table.idx
dk_prj/bin/hpux11/lib.hpux11/resistor.cml
dk_prj/bin/hpux11/lib.hpux11/varactor.cml

```

Note This rather complicated and unnecessary directory structure will be eliminated in a future release when a specific Design Kit directory is created for Verilog-A modules.

Prior to shipping your Design Kit, you can seal the directory by adding another line to the `vamake.spec` file as,

```

USE_CACHE=NO
PREPARE_DIR=NO

```

Now when the simulator boots it will no longer check that the CML files are up to date against the Verilog-A source. If you wish to protect your intellectual property, you can now remove the Verilog-A source files from the Design Kit.

To complete your Design Kit, ensure that the necessary directories and files are available. These required directories and files are described in the table below.

Directory/File	Description
circuit/ael	AEL descriptions of each component
circuit/symbols	Symbol information for each component (it is possible to reference existing component symbols instead)
circuit/bitmaps/pc circuit/bitmaps/unix	Bitmaps for each component
bin/ <i>arch</i>	Compiled model libraries.
de/ael/boot.ael	An AEL file with the load commands for each of the circuit/ael component files.
de/ael/palette.ael	Palette and library loads for each of the components.
design_kit/ads.lib	Information file for the Design Kit loading mechanism to describe the Design Kit.
examples	Place example projects here.
doc	Optional documentation files.

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, the `gcc` compiler version 3.2.3 is used. On these platforms, the system supplied linker and assembler are used. For more information on `gcc`, refer to gcc.gnu.org.

On Microsoft Windows, the *Minimum GNU for Windows (MINGW)* tool suite is used. The MINGW package provides a full compilation environment on Windows. Version 3.2 of MINGW runtime coupled with version 2.3 of the `w32api` is used. The compilation system uses `as.exe` (assembler) and `ld.exe` (linker) from the MINGW `binutils` package version 2.13.90 and the `gcc` compiler, version 3.2.3. For more information on the MINGW tool suite, refer to www.mingw.org.

Compiled Model Library 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,

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

Prior to booting ADS, `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.

Index

A

- aCC compiler tool set, B-1
- ADS Analog RF Simulator (ADSSim), 2-1
- ADSSim, 1-12
- Advanced Design System, 1-1
- AEL, 1-7
- auto loading, 2-1, 2-3

B

- built-in components, 1-1

C

- C compiler, B-1
- cc compiler, B-1
- compilation
 - process, 2-1
 - tools, B-1
- compiled model libraries (CML), 2-1, A-1, B-1
 - CML cache, 2-1
- component palette
 - Devices-Verilog-A, 1-10
- contribution statements, 4-2, 4-8

D

- Data Display, 1-5, 1-6
- ddt operator, 3-1
- design kits, 1-8, A-1
- device
 - behavior, 1-5
 - definition, 2-1
 - mismatch, 1-7
- Devices-Verilog-A palette, 1-10
- diode junction relation, 4-3
- directories
 - custom, 2-2
 - local user, 2-2
 - platform specific, A-1
 - site, 2-2
 - verilog, 1-3, 1-5, 1-6, 2-2, 2-10
- drain current, 1-6
- dynamic linking, 2-10

E

- environment variables
 - AGILENT_CML_BUILDER, B-1

- AGILENT_MODEL_CACHE, 2-6
- USER_VERILOGA_MODEL_PATH, 2-9
- equations, 1-1
 - modifications, 1-5
- error messages, 2-2, 2-10
- examples, 3-1
 - Gummel_Poon SDD model, 4-1
 - PNDIODE UCM, 4-8
 - PSFETV, 1-2, 1-5, 1-6
- explicit loading, 2-1, 2-2

F

- file extensions
 - .va, 2-1
 - .vams, 2-1
- files
 - disciplines.vams, 3-1
 - gumpoon.va, 4-1
 - netlist.log, 4-1
 - parker_skellern.va, 1-6
 - psfetv.va, 1-5, 1-6
 - tutorial_PSFETV.dsn, 1-6
 - vamake.spec, A-1
- functions
 - exp_soft, 4-4

G

- gcc compiler, B-1
- ground, 4-2

H

- HP-UX, B-1

I

- instance parameters, 1-7
- intellectual property, A-1

K

- Kernighan & Ritchie compiler, B-1
- ksh shell, B-1

L

- licenses
 - sim_verilog, 1-12
- load contribution, 4-2
- loading multiple files, 2-2

log, 4-9
log base 10, 4-9

M

macros
 macro constants, 4-5
memory state, 4-5
MINGW tool suite, B-1
models
 GaAs FET, 1-1
 model capability, 1-1
 model cards, 2-1
 nonlinear, 1-8
modules
 module names, 2-2
 Verilog-A, 2-1

N

namespaces, 2-2
natural log, 4-9
netlist, 1-7, 2-1, 2-3
noise, 3-1
 noise contribution, 4-8
nonlinear models, 1-8

O

operators
 ddt, 3-1
 idt, 3-8
 idtmod, 2-8
overriding built-in devices, 2-3
overriding model parameters, 1-7

P

parameters
 device parameters, 2-1
 parameter declaration, 4-9
 parameter macros, 4-9
port values, 4-2
procedural language, 4-1
PSFETV, 1-2

S

sealing the directory, A-2
search path, 1-6, 2-1, 2-3, 2-10
shared libraries, 2-1, B-1
sim_veriloga license, 1-12
simulation environment, 2-1

status message, 2-2, 2-3
status window, 1-4
subcircuit parameters list, 4-2
symbolically defined devices (SDD), 4-1
system values, 4-2

T

temperature, 1-7, 4-9
TIBURONDA_VERILOGA_DESIGN_KIT,
 1-8
time-dependent functionality, 3-1
transient simulations, 4-8

U

user-compiled models (UCM), 2-10, 4-8, A-1

V

variables, 4-5
Verilog-A and UMC devices, 2-10
Verilog-A model search path, 2-1
VerilogA_Load component, 1-6, 2-2

W

warning messages, 2-3, 2-10
weighting functions, 4-2