About Model Development in Verilog-A

About Model Development in Verilog-A

This topic 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 and Verilog-AMS 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 section. 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 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 and Verilog-AMS 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,

```
```

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:

```
```

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 = Is * (exp(V/Vth - Rs * I) - 1)
```

The implementation is shown below.

```
```

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:

```
```

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.

```
```

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.

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:

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:

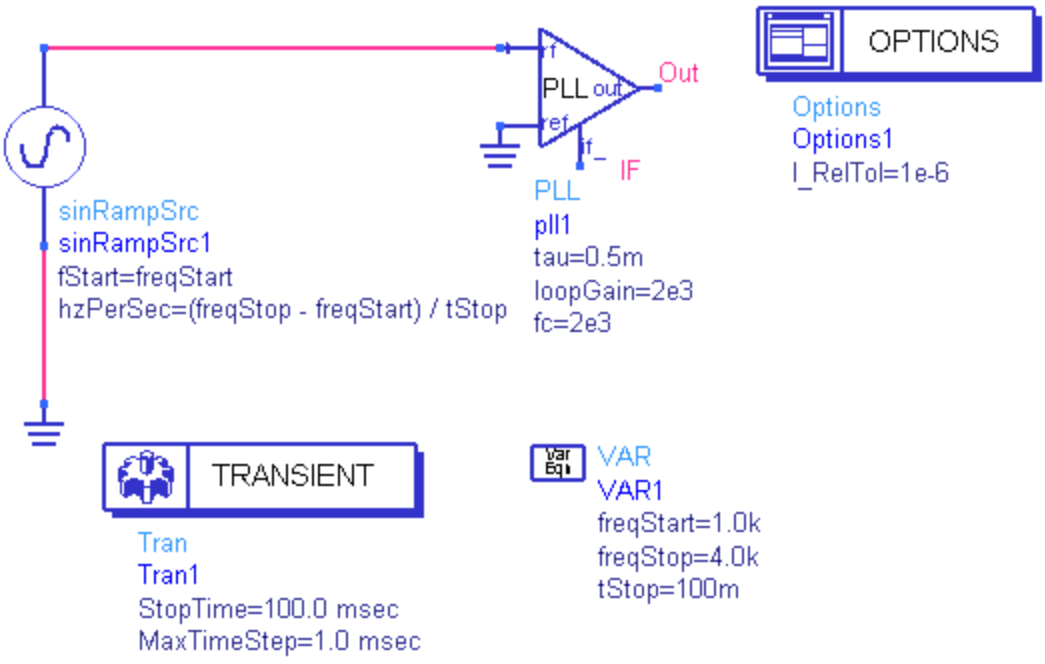
The modules use the keyword `inout` to declare that the ports are both input and output. Some simulators will check consistency of connection of ports (useful when ports are declared input or output only). ADS will not.

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:

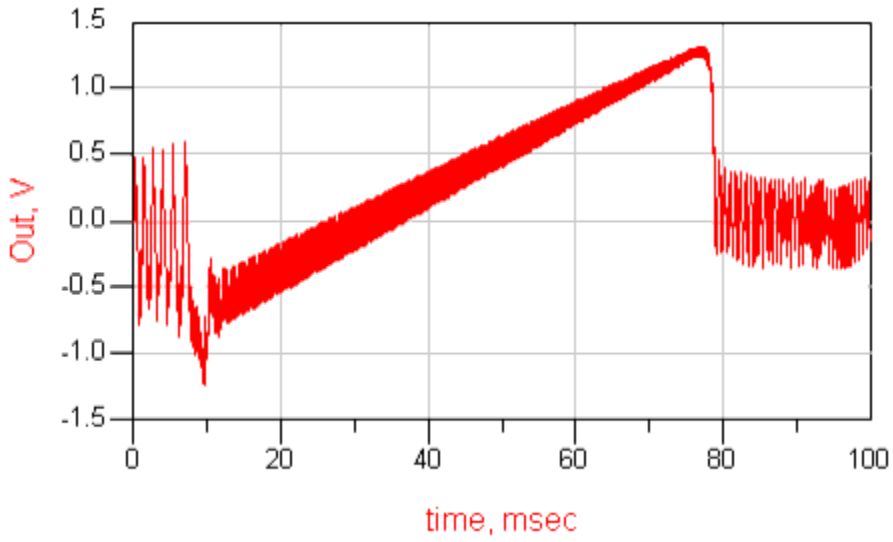
For testing, a swept frequency source is defined:

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



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.



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.

```
```

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.