Hidden State in SpectreRF

Ken Kundert
Designer's Guide Consulting, Inc.

Version 2, 16 May 2019

This document describes the hidden state limitation for Verilog-A models used with SpectreRF, describes a general strategy for avoiding it, and gives some examples. Models for a periodic track-and-hold and a frequency divider are given that are compatible with SpectreRF.

Search Terms Hidden state, Verilog-A, Spectre-RF.

Last updated on June 16, 2024. You can find the most recent version at designers-guide.org. Contact the author via e-mail at ken@designers-guide.com.

1 Shooting Methods

Consider a circuit described by the following set of equations

$$f(v(t), t) = i(v(t)) + \frac{dq(v(t))}{dt} + u(t) = 0 . \quad (1)$$

This equation is capable of modeling any lumped time-invariant nonlinear system, however it is convenient to think of it as being generated from nodal analysis, and so representing a statement of Kirchhoff's Current Law for a circuit containing nonlinear conductors, nonlinear capacitors, and current sources. In this case, $v(t) \in R^N$ is the vector of node voltages, $i(v(t)) \in R^N$ represents the current out of the node from the conductors, $q(v(t))$ represents the charge out of the node from the capacitors, and $u(t)$ represents the current out of the node from the sources.

Traditional SPICE transient analysis solves initial-value problems. In other words, given some initial starting point v_0, transient analysis finds the solution to (1) over the interval $[0,T]$ such that $v(0) = v_0$. A shooting method is an iterative procedure layered on top of transient analysis that is designed to solve boundary-value problems. Boundary-value problems play an important role in RF simulation. For example, assume that (1) is driven with a non-constant T-periodic stimulus. The T-periodic steady state solution is the one that also satisfies the two-point boundary constraint,

$$v(T) - v(0) = 0 . \quad (2)$$

Define the state transition function $\phi_T(v_0, t_0)$ as the solution to (1) at $t_0 + T$ given that it starts at the initial state v_0 at t_0. It is the equivalent of running a transient analysis from t_0 to $t_0 + T$ starting with an initial condition of v_0. In general, one writes

$$v(t_0 + T) = \phi_T(v(t_0), t_0) . \quad (3)$$

Shooting methods combine (2) and (3) into

$$\phi_T(v(0), 0) = v(0) , \quad (4)$$

which is a nonlinear algebraic problem and so Newton methods can be used to solve for $v(0)$. The combination of the Newton and shooting methods are referred to as the shooting-Newton algorithm.

When applying Newton's method to (4), it is necessary to compute both the response of the circuit over one period and the sensitivity of the final state $v(T)$ with respect to changes in the initial state $v(0)$. The sensitivity is used to determine how to correct the initial state to reduce the difference between the initial and final state.

2 Hidden State in SpectreRF

When implementing the shooting-Newton algorithm one must be able to access and manipulate the state of the circuit, in particular, one must be able to set $v(0)$, one must be able to read $v(T)$, and one must be able to compute the sensitivity of $v(T)$ to $v(0)$. Spectre®RF[1] is an RF simulator that is based on shooting-Newton. It provides the usual cast

1. Spectre is a registered trademark of Cadence Design Systems.

of built-in models that everybody has grown to expect from a SPICE-class simulator such as Spectre. And like Spectre, it also provides the user with the ability to write their own models in Verilog-A[2] [1,2]. Unfortunately, the current implementation of Verilog-A sometimes hides state from SpectreRF. SpectreRF is designed to handle the type of state associated with ordinary differential equations. However, Verilog-A introduces a new type of state, the state associated with local state variables, and because SpectreRF is not set up to handle this new type of state, Verilog-A hides it from SpectreRF. Such state is referred to as *hidden state*.

There are two types of hidden state, that which causes SpectreRF to print an error message and terminate, and that which does not. Use of local state variables is likely to cause severe convergence problems, and so is outlawed outright. Whenever such variables are detected, they cause Spectre to terminate before running an RF analysis. Selected functions that result in hidden state are also treated in this manner. They include the *delay* and *z*-domain transfer functions.

The *transition*, *slew*, *cross*, and *last_crossing* functions also suffer from hidden state, though the hidden state generally lasts only for brief periods, and so they are much less likely to produce convergence problems. However, they still can cause problems, and if they are needed it is best to run the simulations in such a way that the functions are not "active" at the beginning or ending of a shooting interval. For example, the transition and slew functions only exhibit hidden state when they are actively transitioning or slew limiting. If you find this happening at a period boundary, try either increasing or decreasing the value of the t_{stab} parameter for the analysis so as to move the period boundary away from the time when the function is active. Of course this is easier, and therefore is less likely to create convergence problems, if the functions transition rapidly. Problems with the *cross* and *last_crossing* functions result if the crossing occurs just after the period boundary. Again, changing t_{stab} generally resolves the problem.

The functions that produce random numbers are generally not suitable for modules destined for use with SpectreRF. These functions include *random*, *dist_uniform*, *dist_normal*, etc. They generally cause two problems. First, one normally stores their value in local state variables, and so creates hidden state. Second, if the random number generators are used to create a stochastic process (a random variable whose value changes with time), then the resulting steady-state behavior of the circuit is generally not periodic or quasiperiodic, which means that the component cannot be used with either PSS or QPSS analyses.

It is not all use of local variables that cause problems, only the use of local state variables. A state variable is different from a simple variable in that on any particular time point its value is used before it is set, which means that its value is retained from a previous point. Consider the simple sample-and-hold model shown in Listing 1. In particular, consider the variables *tstop* and *save*. *tstop* is a simple variable. Notice that within the analog block, it is always set before it is used. Contrast that with *save*, On most time steps it is not set at all. In this situation, it retains its value from a previous point. Thus, it represents state, and it is state that is hidden from SpectreRF.

2. Verilog is a trademark of Cadence Design Systems licensed to Accellera.

LISTING 1 *Sample-and-hold model that exhibits hidden state.*

```
// Ideal Periodic Sample & Hold
// Does not work with SpectreRF (has hidden state)

`include "discipline.h"

module sh(Pout, Nout, Pin, Nin);

electrical Pin, Nin, Pout, Nout;
input Pin, Nin;
output Pout, Nout;
parameter real period=1 from (0:inf);
parameter real tdelay=0 from [0:inf);
parameter real tt=period/100 from (0:inf);
integer n;
real tstop, save;

analog begin

// Determine the sample time
    n = ($abstime – tdelay) / period;
    tstop = n*period + tdelay;

// Sample the input
    @(timer(tstop)) save = V(Pin,Nin);

// Produce output with well-controlled transitions
    V(Pout,Nout) <+ transition(save, tdelay, tt);

end
endmodule
```

3 Avoiding Hidden State

If you have a model that has hidden state and you would like to use it with SpectreRF, you have two choices. Either you can modify the model to eliminate all state that is hidden, or you can reformulate the model in order to expose the state to SpectreRF. Generally, the model uses state for an important reason, and it cannot be eliminated, which generally only leaves the second option. The one common exception is if the model precomputes a set of constants for efficiency reasons. Since the constants are contained in local variables and their values are used at time points well after the values were set, they represent hidden state. This situation is easily remedied by simply recomputing the values on each time point.

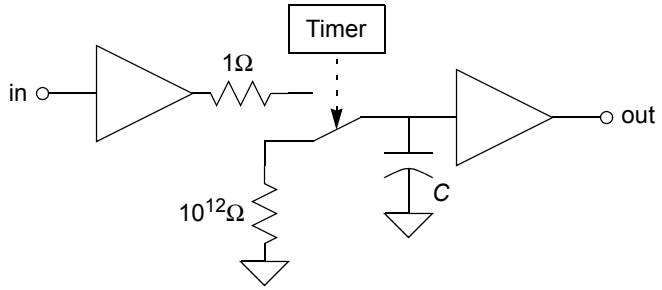
Currently, SpectreRF supports one type of state, the state associated with differential operators. The differential operators include *ddt*, *idt*, *idtmod*, and the Laplace filter functions. Thus, if you wish to reformulate a model to avoid hidden state, and you cannot eliminate the state altogether, you must reformulate it so that state is maintained by one of these operators. Unfortunately, that is generally not a trivial change.

Several examples of how to reformulate models to avoid hidden state problems are given next.

4 Track-and-Hold Model

Since the circuits themselves do not have access to local state variables and other such things, if you build a model by following the basic architecture of the circuit you generally avoid hidden state problems. This approach is taken to build a track and hold model that does not contain hidden state. It is shown in Figure 1 and given in Listing 2. In this

FIGURE 1 *Block diagram of the track-and-hold model of Listing 2.*



case, one builds the track and hold by combining a periodically operated switch with a capacitor. Ideal unity gain buffers are added front and rear to avoid problems with loading. In the spirit of "if you can't fix it, feature it," this model includes effects that would be difficult to incorporate in simpler models, such as droop and a finite aperture time and time constant. Unfortunately, the effects are inherent in the model and even if they are determined to be of no interest they cannot be removed.

5 Frequency Divider Model

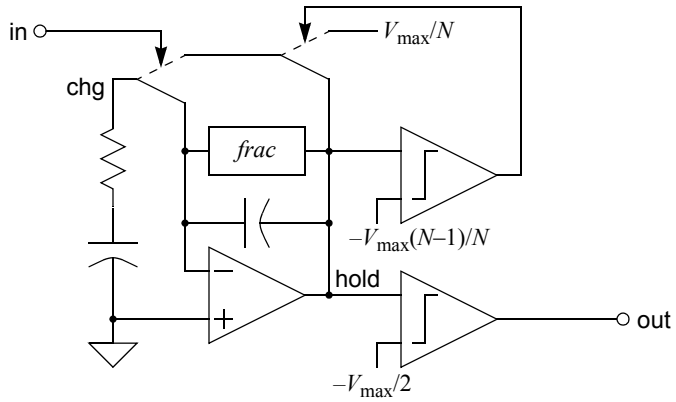
Sometimes following the traditional architecture for implementing a circuit results in a behavioral model that is too expensive. This is the case with frequency dividers, where using binary logic circuits to perform the division generally requires one state variable for every factor of 2 in the division ratio. In these cases, consider taking a non-traditional circuit approach. That was done for the model of a frequency counter shown in Figure 2 and given in Listing 3. This is a switch-capacitor circuit that dumps small packets of charge onto an integrating capacitor and outputs a pulse and resets the capacitor when the voltage on the capacitor gets to a preset level. The size of the charge packet is set to assure that it takes exactly *N* packets to reach the threshold. One packet is transferred to the capacitor for each transition of the input signal in the chosen direction. Thus, one transition occurs on the output for every *N* transitions of the input.

This approach works well for small divide ratios, but for large divide ratios the charge packets become so small that errors can creep in a contaminate the count, especially since the errors accumulate over the entire count. This problem can be reduced by tightening the simulator tolerances, but at the expense of longer simulation times. This model reduces this problem by employing a unique discrete-level feedback around the integrator. This effectively forces the capacitor voltage to always fall at discrete levels. Thus, after each input transition, the any errors that are under the size of a packet are discarded. This greatly increases the range of the divider. However, if the divide ratio is made large enough (> 1000 with default tolerances) then the errors that occur at each

LISTING 2 *A track-and-hold model that does not exhibit hidden state.*

```
// Periodic Track & Hold

// Works with SpectreRF (has no hidden state)
// Almost ideal ...
//    Has buffered input and output (infinite input Z, zero output Z)
//    Exhibits no offset or distortion errors
//    Only nonideality is finite aperture time and very small amount of droop

`include "discipline.h"
`include "constants.h"

module th (Pout, Nout, Pin, Nin);

electrical Pin, Nin, Pout, Nout;
input Pin, Nin;
output Pout, Nout;
parameter real period=1 from (0:inf);
parameter real tdelay=0 from [0:inf);
parameter real aperture=period/100 from (0:period/2);
parameter real tc=aperture/10 from (0:aperture);
integer n;
real tstart, tstop;
electrical hold;

analog begin

// Determine the point where the aperture begins
    n = ($abstime – tdelay + aperture) / period + 0.5;
    tstart = n*period + tdelay – aperture;
    @(timer(tstart));

// Determine the time where the aperture ends
    n = ($abstime – tdelay) / period + 0.0;
    tstop = n*period + tdelay;
    @(timer(tstop));

// Implement switch with effective series resistance of 1 Ohm
    if (($abstime > tstop – aperture) && ($abstime <= tstop))
      I(hold) <+ V(hold) – V(Pin, Nin);
    else
      I(hold) <+ 1.0e–12 * V(hold);

// Implement capacitor with an effective capacitance of tc
    I(hold) <+ tc * ddt(V(hold));

// Buffer output
    V(Pout, Nout) <+ V(hold);

// Control time step tightly during aperture and loosely otherwise
    if (($abstime >= tstop – aperture) && ($abstime < tstop)) begin
      $bound_step(tc);
    end else begin
      $bound_step(period/5);
    end

  end
endmodule
```

FIGURE 2 *Schematic diagram of divider model.*



input transition can become larger than the size of a packet, which corrupts the count even with the discrete-level feedback in place.

Listing 4 is a test model for the divider. It counts the number of input transitions per output transition and prints the number to standard out. It is used simply to confirm that the divider is working properly. It contains hidden-state and so cannot be used with SpectreRF. Listing 5 is the top-level Spectre netlist for the divider test circuit. The waveforms produced by the test circuit are shown in Figure 3.

6 D Flip Flop Model

Listing 6 and Listing 7 show D flip flop models whose variables do not exhibit hidden state.

7 Small-Signal Propagation

When writing highly abstract model of circuits that exhibit thresholding or sampling it is easy to create models that do not properly model the propagation of small-signals. For example, while the track and hold model of Listing 2 models small-signal propagation, the divider and flip flop models o Listing 3, Listing 6, and Listing 7 do not. The difference between the models is that the track and hold has a interval of time where the there is an continuous signal path through every component along the signal path. The track and hold has a switch that often blocks the propagation of the signal from input to output, but at least part of the time in every cycle the switch is closed allowing the signal to get through for a time. In some circuits, such as switched capacitor filters, there are multiple switches that are arranged such that at no time can the signal propagate directly from the input to the output. Rather a switch at the input connects the input signal to a capacitor that stores its value, and then a second switch connects the capacitor to the output. Both switches are never closed at the same time, so there is no direct signal transmission from input to output. But as long as the circuit is modeled such that the two switches are closed for a finite interval of time each cycle, the signal propagation will be modeled correctly even though the intervals do not overlap.

LISTING 3 *Verilog-A model of a frequency divider with no hidden state.*

```
`include "discipline.h"

// Divide by N counter

// Contains no hidden state, and so it works with SpectreRF
// Will fail if N is large, say greater than 1000. Can tighten tolerances
// to increase the range of N, or you can break N into integer factors and
// implement the divide by N with several counters in series.

module divideByN(pout, nout, pin, nin);
output pout, nout;
input pin, nin;
electrical pin, nin, pout, nout;
parameter integer n = 2 from [2:inf);              // divide ratio
parameter integer nhi = 1 from [1:n-1];            // number of hi counts per output pulse
parameter integer dir = 1 from [–1:1] exclude 0;// +1 for rising edge triggered
                                                   // –1 for falling edge triggered
parameter real tt=0.01;                            // output transition time
parameter real vdd=5, vss=0;                       // defines output high and low levels
parameter real thresh=(vdd+vss)/2;                 // input threshold is at midpoint
electrical vg, chg, hold;
real vmax, g, vchg;
integer out, count;

analog begin
    vmax = max(abs(vdd),abs(vss));
    g = 5*ln(n);
    @(cross(V(pin,nin) – thresh, 0));

    // hold capacitor with capacitance tt
    I(vg,hold) <+ tt*ddt(V(vg,hold));

    // clamp hold capacitor during dc analysis
    if (analysis("static"))
        I(vg,hold) <+ V(vg,hold);

    // ideal op amp
    V(hold): V(vg) == 0;

    // titrating capacitor with capacitance tt
    I(chg) <+ tt*ddt(V(chg));

    // titrating capacitor is precharged when input is high
    // if V(hold) is in range,
    //    precharge to vmax/n volts to increment hold cap one level,
    // else
    //    precharge with equal but opposite the charge on the hold cap
    //    in order to reset hold cap to zero
    if ((V(hold) < vmax*(1.5/n–1)) && (dir*V(pin,nin) < dir*thresh)) begin
      vchg = V(hold);
    end else begin
      vchg = vmax/n;
    end

    // determine the count
    count = –n*V(hold)/vmax;
```

continued on next page

LISTING 3 *Verilog-A model of a frequency divider with no hidden state.*

continued from previous page

```
    // charge or discharge when input is high through g Siemen switches
    if (dir*V(pin,nin) > dir*thresh) begin
        // dump titrating charge onto hold cap
        I(vg,chg) <+ g*V(vg,chg);
    end else begin
        // precharge titrating capacitor
        I(chg) <+ g*(V(chg) – vchg);
        // use discrete-level feedback to correct for any errors in hold charge
        I(vg) <+ –n*V(hold)/vmax – count;
    end

    // output pulse when V(in) is high and V(hold) below threshold
    out = count >= (n – nhi);
    V(pout,nout) <+ transition(vss + out*(vdd–vss), 0.0, tt, tt);

end
endmodule
```

LISTING 4 *Test module for the divider.*

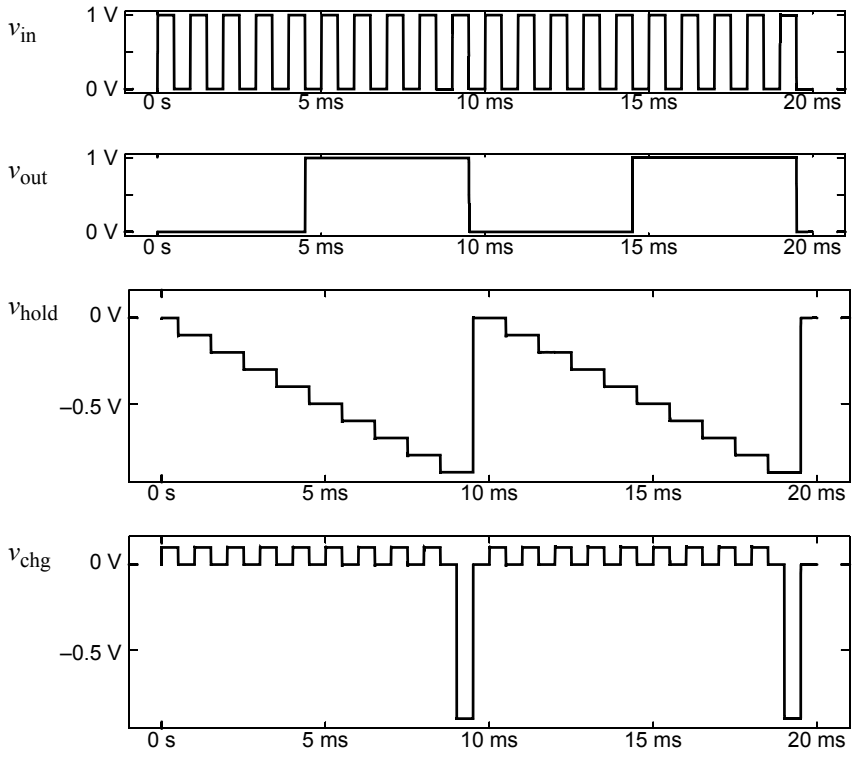
```
`include "discipline.h"

// Divider Test Module

// connect in to input of divider and reset to output of divider.
// set parameter values in the same way as on the divider.
// this module has hidden state and so cannot be used with SpectreRF.

module counter(reset, in);
input in, reset;
electrical in, reset;
parameter real vdd=5, vss=0;
parameter real thresh=(vdd+vss)/2;
parameter integer dir = 1 from [–1:1] exclude 0;
integer n;

analog begin
    @(cross(V(in) – thresh, dir))
        n = n + 1;
    @(cross(V(reset) – thresh, dir)) begin
        $strobe("count = %d\n", n);
        n = 0;
    end

end
endmodule
```

An alternate to the given track and hold model would be to use an ideal sample and hold operation using code like this:

LISTING 5 *Spectre netlist used to test the frequency divider.*

```
// Divide by N test circuit

simulator lang=spectre
ahdl_include "divider.va"

parameters Vdd=1.0
parameters Fin=1kHz
parameters Ratio=10

Vin (in 0) vsource type=pulse val0=0 val1=Vdd period=1/Fin
Div1 (out 0 in 0) divideByN n=Ratio vdd=Vdd tt=0.01/Fin nhi=Ratio/2 dir=-1
// Check (out in) counter vdd=Vdd dir=–1// Comment out if using PSS analysis

setSave options save=allpub
// setReltol options reltol=1e–6          // Only needed for very large divide ratios
save Div1:out Div1:count Div1.hold Div1.chg

pulseResp tran stop=5*Ratio/Fin
pulseSSResp pss period=2*Ratio/Fin     // Must comment out counter for PSS analysis
```

FIGURE 3 *Waveforms from the test circuit with N = 10.*



LISTING 6 *First D flip flop model.*

```
// basic d flip flop
module dff (clk, d, q);

input clk, d; output q;
voltage clk, d, q;
parameter real v0=0;
parameter real v1=1 from (v0:inf);
parameter integer dir=1 from [-1:1] exclude 0;
parameter real td=0 from [0:inf);
parameter real tt=0 from [0:inf);
integer actNow, out;
real thresh;

analog begin
    thresh = (v0+v1)/2;
    actNow = 0;
    @(initial_step or cross(V(clk) - thresh, dir))
        actNow = 1;
    out = idt(0, V(d) > thresh, actNow);
    V(q) <+ transition(out ? v1 : v0, td, tt);
end
endmodule
```

```
analog begin
    @(timer(tstart, T))
        state = V(in);
    V(out) <+ transition(state, 0, 100n);
end
```

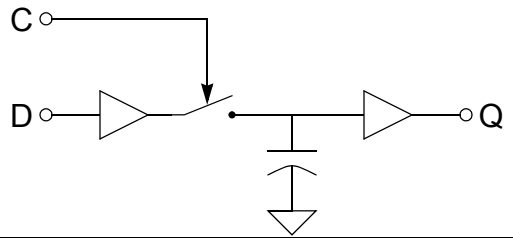
In such a model there is never an interval where the signal can pass through the switch, and so this model does not include small-signal propagation. To understand this it is important to recognize that the simulation breaks the analysis into two steps. The first is a large signal analysis. During this phase the small signal are not present. The circuit is simulated over an entire period and the circuit linearized about that periodically varying operating point. It is the second phase where the small signals are applied. This is the phase that determines whether there will be any small-signal propagation. For small-signal propagation to occur there must be an interval of time involving at least one time point where there is continuous transmission through the switch. In this abstract sample and hold model there is one time point (the sample point) where *state* varies continuously with *V*(*in*), but the *transition* function blocks the small signals. The output of the *transition* function is determined solely by the values present in the large signal analysis.

Figure 4 how a D flip flop could be modeled in such a way that it models small-signal propagation. As such, a model such as this could be included in small signal analysis. For ex mp le, such a model could be used in a frequency synthesizer when simulating phase noise of the synthesizer.

LISTING 7 *Second D flip flop.*

```
// d flip flop with reset (reset is active high) and parameterizable initial state
module dff2 (clk, d, reset, q, qb);
input clk, d, reset; output q, qb;
voltage clk, d, reset, q, qb;
parameter real v0=0;
parameter real v1=1 from (v0:inf);
parameter integer dir=1 from [-1:1] exclude 0;
parameter real td=0 from [0:inf);
parameter real tt=0 from [0:inf);
parameter integer init_state=0 from [0:1];
integer actNow, out, state;
real thresh;

analog begin
    thresh = (v0+v1)/2;
    actNow = 0;
    @(cross(V(clk) - thresh, dir) or cross(V(reset) - thresh, +1)) begin
        actNow = 1;
        state = (V(d) > thresh) && (V(reset) < thresh);
    end
    @(initial_step) begin
        actNow = 1;
        state = init_state;
    end
    out = idt(0, state, actNow);
    V(q) <+ transition(out ? v1 : v0, td, tt);
    V(qb) <+ v0 + v1 - V(q);
end
endmodule
```

FIGURE 4 *D flip flop that includes small-signal propagation as long as the switch model is continuous.*



8 Suppressing Hidden State Error Messages

SpectreRF is proactive about identifying Verilog-A variables that represent hidden state. When it identifies such a variable, it prints an error message and terminates the simulation. The reason being that hidden state variable often cause otherwise mysterious convergence problems. However, in some cases the values of the variables do not actually change during the RF analysis, in which case the variable would not cause convergence problems. Often these particular variables are found in modules used for stimulus or measurement.

In these cases you can mark either the variable, or the module that contains the variable, as safe. To mark a variable as safe, add the *ignore_state* attribute to the variable. To mark all the variables in a module as safe, add the *instrument_module* attribute to the module. Doing so suppresses the error message. It is important to understand that marking the variables in this way in no way addresses the convergence issues that might be caused by these variables. Any variable that you mark as safe must be constant or periodic by the time the RF analysis starts (such as a seed variable) or must not affect the behavior of the circuit in any significant manner (such as a measurement value). Otherwise you will have convergence issues.

Listing 8 shows a module that contains two variables that exhibit hidden state, *period* and *freq*. The variables are marked as being safe in two different ways even though either way would work and both are not needed. These variables are considered safe because their values do not affect the behavior of the circuit.

LISTING 8 *A model that contains hidden state variables that are marked as safe.*

```
// Frequency meter
`include "disciplines.vams"

(*instrument_module*)
module measure_periods(in);
    parameter real thresh=0;    // threshold (V)
    parameter integer dir = 1 from [-1:1] exclude 0;   // 1 for rising edges, -1 for falling
    input in;
    voltage in;
    integer timing;
    real (*ignore_state*) t0, t, period, freq;

    analog begin
        t = last_crossing(V(in) - thresh, dir);
        @(cross(V(in) - thresh, dir)) begin
            if (timing) begin
                period = t - t0;
                freq = 1/period;
                $strobe("period = %es (measured at %es).\n", t - t0, $abstime);
            end
            t0 = t;
            timing = 1;
        end
    end
endmodule
```

9 Conclusion

The Verilog-A hidden state limitations of SpectreRF are a drag, no question about it. However, they can usually be overcome using, of all things, design knowledge. Generally behavioral models that closely follow the structure of the circuit they are modeling are less efficient than more abstract models, and so such a practice is discouraged. In this case it is generally an advantage because it results in models that are less likely to contain hidden state. And to make things more interesting, the normal design trade-offs are much different when your only desire is to write a behavioral model. Thus the cre-

ative designers that take innovative approaches to modeling are generally rewarded with interesting and practical models.

9.1 If You Have Questions

If you have questions about what you have just read, feel free to post them on the *Forum* section of *The Designer's Guide Community* website. Do so by going to *www.designers-guide.org/Forum*.

References

[1] Kenneth S. Kundert. *The Designer's Guide to Verilog-AMS*. Kluwer Academic Publishers, 2004.

[2] *Verilog-AMS Language Reference Manual: Analog & Mixed-Signal Extensions to Verilog HDL*, version 2.1. Accellera, January 20, 2003. Available from *www.accellera.com*. An abridged version is available from *verilogams.com* or *designers-guide.org/verilog-ams*.