Designer's Guide Consulting
Analog Verification

ANALOG VERIFICATION NEWSLETTER
NUMBER 8, JULY 2009

Contents

Greetings

Dear friends and colleagues,

For this newsletter, rather than focus on an over-arching analog verification concept, we thought we'd write on some of the things that often go wrong when writing models and regression tests in Verilog-AMS. We'd also like to announce two classes that we'll be teaching. The first is our regular 4 day *Analog Verification* (AV) training class. It will be held 29 September 2009 to 2 October 2009 in Mountain View, CA. The target audience for this class is those who anticipate practicing AV, those who want to improve on their AV skills, or those who want an in depth, hands-on feel for what AV is. This is contrasted with our a newly created 2 day *Introduction to Analog Verification* class to be held 20 October 2009 and 21 October 2009. The target audience of this class is those who will be working with AV engineers such as analog design engineers, digital verification engineers, design leads and managers; and for those who want an in depth introduction to AV.

There is no doubt that the economy is bad. We think that now is a great time to brush up on your skills or learn new skills. For this reason, we are offering our classes at half off if you are unemployed. This offer is made on a space-available basis and at our sole discretion. As we will most likely be unable to confirm a spot until 1 or 2 weeks before the class, we suggest that only those unemployed engineers living near Silicon Valley should sign up as it may be difficult to make travel plans on such short notice. Please send us e-mail if you are interested.

Besides offering open classes, both of these classes are also available to be taught on-site at companies throughout the world. Please contact us for scheduling. We'd also like remind everyone that the main focus of our company is to offer consulting services to help your company transition to applying analog verification. We also offer modeling services. Details on our on-site classes and our services can be found at *www.designers-guide.com/services*.

As always, feedback is greatly appreciated.

Sincerely,

Henry Chang and *Ken Kundert*

Announcements

4 Day Training Class: Analog Verification

29 September — 2 October, 2009 in Mountain View, California

This challenging four day course provides participants with the tools they need to take on the task of verifying complex analog, RF, and mixed-signal integrated circuits. It combines lecture with a substantial amount of time in the lab to teach the overall analog verification process. You will learn how to develop a verification plan, functional models of analog blocks, regression tests for those models, and a fully verified Verilog model for the entire analog portion of the design for use in chip-level verification.

Target Audience. The class is intended for anyone who would benefit from a working knowledge of analog verification. These include: analog verification engineers, analog designers, analog design leads, and digital verification engineers and CAD engineers who meet the prerequisites.

Instructors. Ken Kundert and Henry Chang.

Prerequisites. Students should have a working knowledge of Verilog-A, analog circuits, and the Cadence design environment. It is also helpful to have gone through Verilog-AMS training. The better prepared you are, the more you will get from the class.

Cost. The class will be held from 29 September to 2 October in Mountain View, CA. The price is $2350 until August 28, at which time it becomes $2700.

For more information or to sign up, visit *www.designers-guide.com/classes*.

2 Day Training Class: Introduction to Analog Verification

20 October — 21 October, 2009 in Mountain View, California

This two day course provides an in depth introduction to the verification of complex analog, RF, and mixed-signal integrated circuits. It combines lecture with labs to illustrate the concepts in analog verification. You will learn the principles of analog verification and the basics so that you can effectively interact with analog verification engineers such as working with them to develop the verification plan, understanding analog verification terminology, and gaining some ability to read the models and regression tests they create.

Target Audience. Those who want to get a solid introduction to analog verification — design managers, design leads, digital verification engineers, analog designers, CAD engineers, and anyone who needs to work with analog verification engineers, as well as those considering becoming analog verification engineers.

Instructors. Ken Kundert and Henry Chang.

Prerequisites. Students should have some knowledge of analog circuits. Knowing how to use the Cadence design environment is a plus. Experience in modeling with Verilog, Verilog-A, and Verilog-AMS is also a plus, but it is not necessary. The better prepared you are, the more you will get from the class.

Cost. The class will be held from 20 October— 21 October in Mountain View, CA. The price is $1410 until 18 September, at which time it becomes $1620.

Please also feel free to call us or to send us e-mail if you are unsure which class is the most appropriate for you.

For more information or to sign up, visit *www.designers-guide.com/classes*.

A Few Gotcha's When Using Verilog-AMS

By Henry Chang and Ken Kundert

Between our 4 and 2 day analog verification classes, we've now taught analog verification class to over 150 people. We've observed many common mistakes as people work on the lab exercises. Though our lectures and lecture notes provide sufficient information to avoid these mistakes, there's a lot of detail when it comes to using Verilog, Verilog-A, and Verilog-AMS, and it's difficult to get it all at once. So although there might be a steep learning curve, the good news is that once you understand the basics and the key gotcha's, you'll be in good shape.

To make this article interesting for those who are already experts at Verilog-AMS, we're going to pose these gotcha's in the form of a quiz. Please feel free to send us e-mail with how you did. If we get enough replies, we'll publish the statistics in our next newsletter. And for those of you who have taken our 4 day class, one of these gotcha's is new.

The examples we present are complete models, and should work in any Verilog-AMS simulator. Often, "we," the collective of all of us who have ever used a circuit simulator, blame the simulator for unexpected results. In these cases, we assert that these gotcha's are not simulation issues, but modeling issues. To support this assertion, we've tried these examples on more than one Verilog-AMS simulator, and the same issues are exhibited.

Question #1

Listing 1 shows a testbench that generates a 1 MHz sine wave voltage on the electrical node, *dut_in*. No device-under-test (DUT) is yet instantiated. For the purposes of this example, we're going to suppose that we're in the initial stages of developing a testbench. We begin without a load just to make sure our testbench is working as we expect. From the testbench, what is desired is a sine wave at 1 MHz that simulates for approximately 10 periods.

Question: Why will this not give what is desired? The results from two simulators are shown in Figure 1.

Hint: There's nothing wrong with the code that's there. In fact, from looking at the waveforms, one can observe that the simulators are doing exactly as they are told. One line can be added to the model to give what is desired.

```
`include "disciplines.vams"
`include "constants.vams"
`timescale 1s/1ns

module testbench ();
real freq;
initial freq = 1M;
electrical dut_in;

initial begin
    #10u $finish;
end

analog begin
    V(dut_in) <+ sin(2*`M_PI*freq*$abstime);
end
endmodule
```

Listing 1: Sine wave source

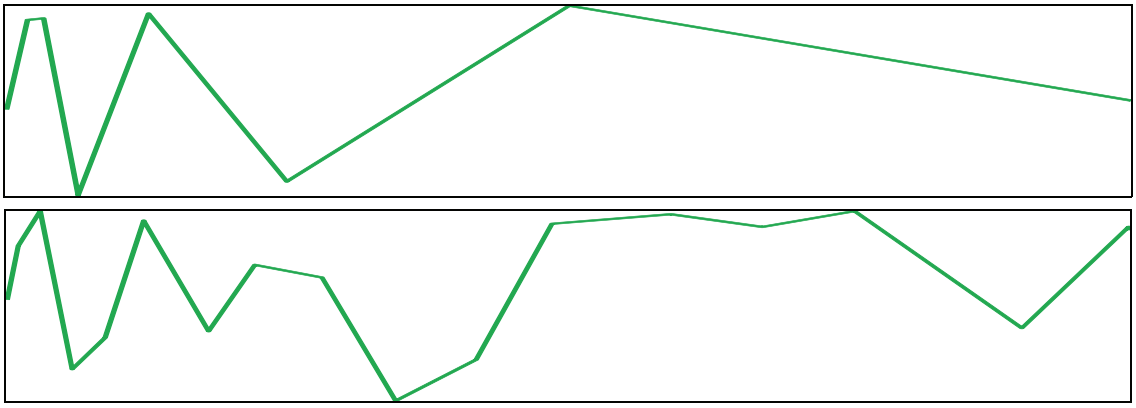

Figure 1: Output from two different Verilog-AMS simulators

Question #2

Here is a model (Listing 2) and a testbench (Listing 3). The model is of an 8 bit DAC. The intention is that it produce an output proportional to the input from 0V to 1V, so that if *in* is 0, it is 0V, and if *in* is 255, the output, *out*, is 1V. The intent of the testbench is that it instantiates the DUT and goes through each input code of the DAC and then terminates the simulation. If this were one of our actual testbenches, we'd have self checking code after the "#1u" statement to verify that the output is what is expected. However, suppose, in this case, we just want to view the output in a waveform viewer.

Question: Why will this not do as we expect? How can we remedy the situation?

Hint: The issue is in the testbench. Our digital readers should be able to easily spot the issue.

```
`include "disciplines.vams"
`timescale 1s/1ns

module dac (in, out);
parameter real max_out = 1.0;
input [7:0] in;
output out;
electrical out;

analog begin
    V(out) <+ transition(in*max_out/255, 0, 100n);
end
endmodule
```

Listing 2: Simple DAC model

Question #3

Listing 4 has a model of a variable load. The intent is that if *setting* is 1, that this module models a 150 Ω resistor. If the *setting* is 0, we want the module to model a 250 Ω resistor. The testbench is shown in Listing 5. It goes through both of the settings and displays the output current. The *in* voltage is set to 1V. To be clear from which branch we're observing the current, we create a branch from *in* to ground called *supply*. The branch current, *I*(*supply*), is, therefore, the current flowing through the voltage source. Thus, we'd expect that the current, *I*(*supply*), is negative, so that the current flowing through the load (from *in* to ground) is positive.

```
`include "disciplines.vams"
`timescale 1s/1ns

module testbench ();
reg [7:0] dac_in;
electrical dac_out;

dac DUT (dac_in, dac_out);

initial begin
      for (dac_in = 0; dac_in < 256; dac_in = dac_in + 1)
            #1u;
      $finish;
end
endmodule
```

Listing 3: Simple DAC Testbench and Regression Test

```
`include "disciplines.vams"
`timescale 1s/1ns

module var_load (setting, in);
input setting;
electrical in;

analog begin
      V(in) <+ transition(100*(~setting + 1.5), 0, 100n)*I(in);
end
endmodule
```

Listing 4: Model of a Variable Load

```
`include "disciplines.vams"
`timescale 1s/1ns

module testbench();
reg setting;
electrical in;
var_load DUT (setting, in);
branch (in) supply;

initial begin
      setting = 0;
      #1u $display ("Setting=%b: Current is %f", setting, I(supply));
      setting = 1;
      #1u $display ("Setting=%b: Current is %f", setting, I(supply));
      #1u $finish;
end

analog V(supply) <+ 1;
endmodule
```

Listing 5: Testbench for the Variable Load

Question: What are the currents we should achieve for the two settings based on our description of what we want? Why do we not achieve these results? How do we remedy the situation?

Hint: There is an error in the model, and the resulting error in the current is not subtle. In fact, if we could actually implement this model with only the inputs as shown, we'd never have to worry about finding new sources of energy.

Answer to Question #1

The sine wave does not look like a sine wave because the simulator is taking time steps that are too large. However, the results are "correct" in the sense that the values computed by the simulator do fall precisely on the desired sine wave. However, the very low density of time points makes it very difficult to correctly interpret the simulation results. In Figure 1 the time points are where the slope of the output changes. Recall that usually by default, the waveform viewer plots a straight line between the results at each time point. Given nothing else in the circuit or no other constraints on the simulator, the simulator's goal is to choose as few time points as possible while satisfying the equations presented, so that the simulation can complete in the shortest time possible. So, had there been more to this circuit, where more equations had to be satisfied, it is *likely* that more timesteps would have been required and the sine wave would have been traced more faithfully. "Likely", however, is not a word we like to use in verification.

What's the solution? There are actually two answers depending on what the desired result is. The first and more critical is that we want to ensure that the rest of the circuit sees the sine wave. For example, had the time steps been chosen at exactly the time at which the sine wave crosses 0 (at half period intervals of the sine wave), the output of the sine wave source would look like a straight line, and even with a complex circuit loading the sine wave source, it would appear as if we had a DC voltage source at 0V. To address this, we use the $bound_step command as shown in Listing 6.

```
analog begin
    V(out) <+ sin(2*`M_PI*freq*$abstime);
$bound_step (0.3/freq);
end
```

Listing 6: Use of $bound_step

$bound_step forces the simulator to take the next time step at no greater than the current time plus the argument that is passed to the $bound_step command. To ensure that the sine wave is "visible," we set $bound_step to be approximately 1/3 of the period of the sine wave. In this way, it is impossible for the time steps to be at the zero crossings, i.e. at 1/2 the period.

If you were to add this line and run the simulation, you'd likely still be unhappy with the result. Although it would look better, having only 3 or 4 points per period rendering a sine wave is not very "pretty". To make a "pretty" sine wave, multiplying the period by 0.03 instead of 0.3 should be sufficient.

You might wonder why you've never encountered this problem before even though you've been using sine wave sources and circuit simulators for most of your career and in school. That's because most simulators have the concept of $bound_step built into their sources. Also, with transistors, there is a lot of dynamic behavior causing the circuit simulator to take many time steps. Recall that it doesn't matter what causes a time step in a circuit, once a time step is taken, every equation in the circuit is evaluated and attempted to be made true within the error tolerances.

If you've never used $bound_step before and decide to begin applying it as a result of reading this article, the old adage of "a little knowledge is a dangerous thing" applies. $bound_step can only add time steps, which means that $bound_step can *only* slow down the simulation. There are a few coding guidelines we apply when using $bound_step. The first is that $bound_step should only be applied in the model where the issue may lie. Applied elsewhere, and someone may include a $bound_step when it isn't necessary. The second is that often sources can be turned on and off. In the off state, make sure $bound_step is not being applied or set to a very large value, such as 1s. Otherwise, you are needlessly slowing down the simulation. Finally, only use $bound_step when necessary, and try to use the highest value possible. The most dangerous scenario is using someone else's model without knowing that a $bound_step is in their code. Usually, this leads to very difficult to debug slow simulations. At the chip level where there may be dozens to hundreds of models when you include all of the digital, it's very difficult to figure out what's causing the time step to be small.

Answer to Question #2

The result of simulating this model and testbench is that the simulation never terminates, i.e. execution never reaches the $finish statement. The issue is caused by *dac_in* being an 8 bit register. It can only have values between 0 and 255, i.e. if you add 1 to 255, you get 0. As a result, the "*dac_in < 256*" in the *for* loop is always true, and therefore, this loop never terminates.

Our coding guideline when it comes to *for* loops is to always use an integer as an index variable, and the set input to a model or circuit to that index variable. This is shown in Listing 7.

```
integer i;

initial begin
    for (i = 0; i < 256; i = i + 1) begin
        dac_in = i;
        #1;
    end
    $finish;
end
```

Listing 7: Using an index variable instead of the register in the loop

Answer to Question #3:

What we want is for the "supply" current flowing *into* the supply node to be –6.7mA for setting 0 and –4mA for setting 1. What we achieve is:

```
Setting=0: Current is –0.020000
Setting=1: Current is 0.020000
```

Current direction is often confusing, but to have two settings produce currents in opposite directions seems to imply that in one of the settings, we have a negative resistance. And, in fact, we do. The issue is in our use of the "*~setting*" statement. Recall that the "~" is the bit-wise inversion operator. In the digital section, if *setting* were 1, the result would be 0, and if *setting* were 0, the result would be 1. In Verilog-AMS, however, there is no "*reg*" type in the analog block. When "~" operates on *setting*, *setting* is first converted to a 32 bit two's complement integer. In particular, setting = 0 is converted to 0...0000. The bit-wise inverse of which is 1...1111. The integer result is –1. In the other case, setting = 1 is converted to 0...0001. The bit-wise inverse of which is 1...1110. The integer result is –2. Thus for the equation:

```
100 * (~setting + 1.5)
```

We get 100*(0.5) and 100*(–0.5). One volt across a 50 Ω resistance gives –20mA of current into the source, and across a –50 Ω resistor gives 20mA of current *into* the source. A solution is to use the logical inversion operator "!" rather than the bit-wise inversion operator "~".

```
100 * (!setting + 1.5)
```

However, this is problematic when the value of *setting* is either *X* of *Z*. In this case, the value of the expression becomes unknown. This may result in simulator dependent behavior. One common outcome is that the simulation will stop if an unknown value enters the analog kernel. A better solution, and one that is in line with our coding guidelines, would be to replace the *V*(*in*) contribution statement with:

```
V(in) <+ transition(100*((setting === 0 ? 0 : 1) + 1.5), 0, 100n)*I(in);
```

In this case we assure that an *X* or *Z* on *setting* cannot propagate into the analog kernel. Here, when *setting* is 1'bx, the resistance of the load becomes 250 Ω. In actual models, we'd carefully control what happens on an *X* or *Z*. We may print a message to the log file, so that we're aware this happened. We may also determine that it is natural that as the digital circuitry starts up that there are *X's* and *Z's* on the control bits. In this case, we would give careful thought as to how we want the model to behave. It is for the reason that the analog block undefined issue often does not exhibit itself until chip level simulations when the digital and analog blocks are combined.

Conclusions

For the sake of simplicity, we simplified our models to the point where they do not fully adhere to some of our other coding guidelines. We'll save more of those for another newsletter article, or as an exercise to the reader. If you think you can see other issues with the examples we've presented, feel free to mail them to us.

We've presented three common mistakes — one related to the analog section, one related to the digital, and one related to mixing the two. Being aware of these and using a set of coding guidelines can prevent most common Verilog-AMS mistakes. In our 4 day class, we go over coding guidelines and explain the underlying reasons for why they are necessary.

The mistakes presented in this article largely cause functional issues. There are also gotcha's when it comes to simulation efficiency. We also use coding guidelines to ensure that our models and regression tests are efficient keeping in mind that the models must be fast enough to run for a significant amount of circuit time at the chip level, at least 100's of milliseconds if not 10's of seconds. As we've suggested, a poorly used $bound_step can limit performance. Another example would be a poorly chosen rise and fall time in transition statements. As a default, do you often just write "1n" for the rise and fall time? If the digital clock is only operating at 1 MHz and that is the fastest moving signal, are you aware that this could slow down your simulation by as much as 10x. 1n is also a poor choice if you're operating in the GHz range. Our coding guideline for this is simply that you don't blindly choose rise and fall times in transition statements.