



# Designer's Guide Consulting

## Analog Verification

ANALOG VERIFICATION NEWSLETTER

NUMBER 9, AUGUST 2009

### Contents

- 1 - Announcements
- 2 - The Challenges of Chip Level Simulation

### Greetings

Dear friends and colleagues,

We'd like to remind you of the two classes that we'll be teaching: our regular 4 day *Analog Verification* training class (29 Sep. 2009 to 2 Oct. 2009) and our 2 day *Introduction to Analog Verification* class (20 Oct. 2009 and 21 Oct. 2009). Both will be held in Mountain View, CA. The full course is more appropriate for engineers that will be actually doing analog verification and for those designers that want a solid working knowledge of Verilog-AMS. The second is a shorter class designed for those who will be working with the analog verification engineers. We also offer both of these classes to be taught on-site at companies throughout the world. Please contact us for scheduling. Details on our on-site classes and our consulting services can be found at [www.designers-guide.com/services](http://www.designers-guide.com/services).

In this newsletter, we write on one of the most difficult aspects of analog verification: chip-level simulation; in particular, the challenges you face when attempting chip-level simulation and how to overcome them.

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](http://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](http://www.designers-guide.com/classes).

## **The Challenges of Chip Level Simulation**

*By Henry Chang*

The most difficult but often the most critical aspect of analog verification is running the chip-level simulation. Running chip-level simulations includes watching the chip go through its entire power up sequence, seeing the chip running op codes on an embedded processor which then affects the analog circuitry, and using the chip almost as if it were the manufactured part. Chip-level simulations are usually the culmination of months of work. Prior to chip level simulation, almost all of the time is spent developing block level models, writing regression tests to verify that the models match the implementation, planning for the chip level simulations, and hopefully finding bugs at the block level. Painstaking effort is applied to ensure that models will run quickly in anticipation of the chip level simulations. This includes making decisions as to what constitutes function vs. performance, what to model in the analog vs. event-driven domain, and what analog signals are to be passed as electrical signals vs. discrete analog values in the form of wreals in Verilog-AMS or by using an equivalent method. If set up carefully, seconds to tens of



seconds of transient simulation can be run in a reasonable amount of time, such as an hour. This can be done even when there are clocks present in the tens of megahertz. And with baseband equivalent or phase domain modeling, this can be achieved with signals that are in the gigahertz range on a complex chip such as an RF system where there is RF, analog, RTL, and an embedded processor.

The chip level simulations are critical as the chip level is an area rich in bugs. Common bugs include incorrectly connected bias signals, control signals miswired, and supply lines in wrong supply domains. Although there can be other ways to find these bugs, such as by visual inspection, there are some bugs that can only be found with chip level simulations. An example are certain bugs in the RTL. It is during these simulations that the RTL is first simulated with the analog where the models used for the analog truly match the implementation, and not where the models were created by the digital designers who are generally unaware of all of the assumptions put into the analog blocks. As other benefits, it's certainly satisfying to see that existing tools can provide such capabilities. It's also gratifying to hand off the test setup to a chip lead so that he or she can run experiments on their chip on their own.

Getting the chip-level simulations to work is usually difficult. It often involves a lot of painful and frustrating work. Make no mistake, this is a daunting task in most complex analog, mixed-signal, and RF chips. To get this to work, we apply a bit of art, and an arsenal of experience. In this article, we'll discuss a few of the tactics to get through the process. The tactics are most applicable when using Verilog-AMS models, but the basic principles are applicable even if you are using pure Verilog to model the analog.

### ***Tactic #1: Write models keeping in mind that they will be in chip-level simulations***

It is critical to plan and to expect that you will be doing chip-level simulations. Planning involves ensuring that the models will run quickly making the many trade-off decisions as discussed above. It also includes writing the models somewhat defensively knowing that the models will be used in a different context than that of the block level regression tests. I give this advice with a bit of hesitancy, because what modelers typically do when they model defensively is to focus on smoothing out signals with time expensive mathematical functions. This often doesn't address the primary issues that occur at the chip-level, while certainly slowing down the simulations. Thus, this approach provides no benefit, but does reduce efficiency. So, when I say defensive, what I mean is to be defensive for the situations that actually occur at the chip level and at no time do anything that unnecessarily slows down the simulation. Here are some common issues that occur at the chip level.

The first are X's and Z's from the RTL. In a block-level regression test, it is rare that the regression test will drive an X or Z into the model unless this is purposely done. The first part of the test usually initializes all of the pins of the device under test. However, at the chip-level, the RTL very typically initializes pins to X or Z. Digital designers and digital verification engineers use the X or Z to see what is going on in the hardware they are designing. They make no effort to remove them and usually take advantage of having them. So, one modeling technique we use is to filter digital inputs to remove X's and Z's.

Another source of errors at the chip level is due to current sources not being connected to anything. In a block-level regression test, it is rare that there will not be a sink for every source and vice versa. However, when first working at the chip-level, the design is usually in a state of flux where not all of the current sources are connected, since not all of the blocks have made their way into the top-level. Having an unconnected ideal current source in the model will likely cause the simulation to fail, because there is no current path for the current to flow. To avoid this, some



place a conductance of  $g_{min}$  on every current source. Although this will give a current path, this is usually not a good solution, as an extremely high voltage will result on that node. High voltages can cause the simulator to effectively turn off its error controls if the proper options aren't set. If the simulator has an "alllocal" type option, this can alleviate this issue. As a better option, current sources can be modeled so that if unconnected, the voltage on their output never goes below ground or above  $V_{dd}$ . After all, this is what real current sources do. Take care, however, when doing this not to violate the speed rule by using a needlessly complex math function to achieve this.

Putting assertions on the pins of the models are also an excellent way to catch potential issues. Assertions codify assumptions on the pins of the models. For example, an assertion can be placed on the supply input to verify that the input is within the desired supply voltage range. In practice, we place assertions on as many inputs as make sense. On occasion we write them for outputs.

Remember, the reason for taking the time to do this before chip-level simulations is that it is very difficult to debug anything at the chip level. Netlisting and just starting the simulations take time, and if there is an issue such as a convergence problem, the simulation speed slowdown is magnified at the chip-level. Debugging a single issue often takes an afternoon. It's much better to avoid these problems in the first place.

### ***Tactic #2: Focus on simulation speed***

Simulation speed is of such importance that I suggest debugging speed problems immediately upon seeing a simulation slowdown. In debugging the issue, I usually expect to spend one to two hours and am willing to spend up to a day. This applies at the block and chip level. I define a simulation slowdown as a simulation where the timesteps are smaller than expected or where all of a sudden the simulation slows down by a significant factor. Consider that debugging the issue will only get more expensive later. If simulation times are already slow and more blocks are added, the simulation times can only continue to get worse. Thus, as more blocks are added into the mix, there will only be more stuff to hunt through to figure out where the problem is.

When debugging a slow simulation, a common tactic that I employ is to first determine if there is a relatively uniform small timestep, and then see if the frequency that it corresponds to is a multiple of any of the clocks in the system. Upon associating the timestep with a clock, if the clock was expected to stay in the discrete domain, I look for where it leaked into the analog section. If I expected the clock to be in the analog domain, I consider moving it to the discrete domain. For clocks that leak into the analog section, unexpected placement of a connect module can often be the source of the problem. For example, if the clock is to stay in the discrete domain, there must not be a connect module taking it into the electrical domain. There are situations where a connect module is inserted even though no blocks actually consume the electrical signal generated. This is usually correctable by setting a discrete discipline, e.g. logic, on the pin that was considered to be electrical. An approach that often works is to set the discipline hierarchically from the regression test. Unexpected bidirectional connect module insertion can also be the culprit of this problem and others. Since I generally never expect a bidirectional connect module, I usually use connect rules that do not have bidirectional connect modules. In this way, I am certain that one will not be added. If the elaborator tries to add one, I will get an easy to pin-point error to the module where there is an issue. In most situations, the "inout" in question should be converted to either an input or output.

I also look for math functions in the models. As mentioned earlier, these are often used to make the models more "robust." More often than not, these are used needlessly. Once found, I remove



these unless I can assure myself that having them is the only approach that will work. This habit of extensively using math functions has historical roots. Most probably it came from the Verilog-A world at a time when simulators didn't handle behavioral models well. Circuit simulators have come along way, and now with Verilog-AMS, there's the discrete section where computation can occur with little to no impact on the analog simulator. I suggest re-examining old assumptions on modeling techniques when it comes to the use of these math functions to see if they still hold true in the light of today's simulators and Verilog-AMS.

Finally, I look at the circuit inventory in the simulation log file to see if there's anything unexpected. I count the number of connect modules to see if it matches what I expect. I also look for stray devices or passive components. Often, the schematic is set up in such a way that it's difficult to completely avoid all transistors, but if I see more than a hundred, that's usually a sign to me that something is wrong. It could be as simple as something not being modeled. It could also be that a discussion is required with the design lead to re-arrange the hierarchy in the schematics. Often, a simple change can give large payoffs. For example, if there are a few stray transistors in the top-level schematic, just by putting those transistors in a block can get rid of those. This is usually considered a small change on the part of the designers.

### ***Tactic #3: Pace yourself when following the 5 steps to successful chip-level simulations***

This tactic applies at the point when first trying to start a chip-level simulation. As the fictional character Yoda says, "Do, or do not. There is no 'try.'" Believe that you will succeed! Expect though that it will take time and don't set any unrealistic expectations. No one expects a chip *design* to be done in a week or two. Take comfort in knowing that eight to ten designers for 4 to 6 months have been diligently working to add more and more complexity to the chip. The chip level simulation is, after all, the "moment of truth." It will take time.

Typically, at this stage, the design is still changing at the block and chip-level, the top-level schematic has just been created with little regard for what the analog verification engineer may need, and at the point where the designers have not yet attempted any kind of chip-level simulation. Expect at least a week to get any kind of chip-level simulation working, but more likely two weeks of often frustrating two step forward, one step backward kind of work. Hopefully, there is still one month before tape-out. A month and a half would be better. Prior to this, I advise pushing on any roadblock to get to this stage sooner. We apply different tactics for analog verification when there is a limited amount of time prior to tape out, but that's a different topic.

I break down getting the chip-level simulations started into 5 steps. I consider myself lucky if any of these steps require less than a day, and worry if any step takes longer than three days.

*Step 1: Prep work*

*Step 2: Getting the chip to netlist and compile*

*Step 3: Getting past the elaboration phase*

*Step 4: Simulating past  $t=0$  in a reasonable amount of time*

*Step 5: Simulating a period of time equivalent to 100 to 1000 times the shortest clock period in a few minutes and observing that whatever has been activated in the chip is doing what it is supposed to*

For those not familiar with the simulation terms mentioned, netlisting is the process of converting the schematics to a form that the simulator's compiler can handle. For example, schematics can be converted into a Verilog netlist that in turn gets compiled by a Verilog compiler. Compilation is the process of converting a model or netlist into machine executable code. Elaboration is



the process of “linking” the compiled code together. Simulation is the process of finding the operating point (for the analog) and then computing what the design will do as time advances.

If these 5 steps can be accomplished, then I expect that chip-level verification will go smoothly. Getting through these 5 steps is basically an exercise in debugging. The basic process is to identify each issue one at a time, methodically find a solution, and then move onto the next. Below is a list of some of what it takes to get through each step.

#### *Step 1: Prep work*

Verify that 80% to 100% of the models have been written, especially those that are critical to the start up sequence of the chip, such as a bandgap reference. For the non startup critical models that are not present, prepare empty models for those. By habit, I create an empty model for every modeled block. In the Cadence environment, I might use the view name “empty” for the empty models to distinguish it from the actual models. An example of a non-critical model might be a general purpose ADC, i.e. a block that is not required during the boot-up sequence. For non modeled blocks that are critical, use the schematics. Blocks like reference generators and band-gaps can usually simulate relatively quickly at transistor level. Alternatively, use an early version of the model where, for example, not all of the trim bits have been modeled. It may also be possible to start without the RTL, as some basic analog signal flow can often occur with only a few enable bits active on the analog blocks. These can be set in the testbench, for example, by using Verilog force statements. Creativity pays off here to reduce the amount of effort required to get started.

Prepare a simple top level testbench. Although our general practice is to have Verilog-AMS as the top level model, we’ve encountered issues where some tools cannot handle SystemVerilog instantiated in Verilog-AMS. If you are mixing various languages, the safest approach is to have Verilog on top, and have it instantiate the chip and testbench. The Verilog on top can be in the form of a Verilog module or a schematic.

#### *Step 2: Getting the chip to netlist and compile*

I group netlisting and compiling together since compilation usually isn’t an issue. The models tend to compile without issues since they were tested by themselves prior to attempting a chip-level simulation. The schematics also tend to compile fine. The challenge is usually in netlisting.

A big issue here is that the design is in a state of flux. Nothing can be done about that. If any amount of verification is to be finished before the design tapes out, we necessarily have to start chip-level verification while the design is in flux. Here, the focus is to try to get a relatively stable snapshot of the design. See if the revision control system can help. Work with the lead designers to try to get a consistent view of the entire design. Time will likely have to be spent working with the lead designer to debug the top-level netlist, and maybe the symbol for the RTL as the symbol is often manually created and has to be changed whenever the RTL changes.

When running in the Cadence environment, a common problem is when the schematics have not been “checked and saved.” If all of the schematics needed have not been “checked and saved,” netlisting will fail. Correcting this may be as simple as opening the schematics and doing a save, but often this cannot be done for a couple of reasons. As the verification engineer, we often do not have write access to the schematics. There might also be a revision control system in place where others have checked out the blocks that we need. Resolving this usually requires a quick e-mail or phone call to the schematic’s owner and an annoying unproductive wait time. There might also be some database inconsistencies such as a property file or symbol mismatch. My advice is to work through these issues in the office at a time when you have quick access to the schematic owners. Warn people ahead of time that you are working on chip-level simulations



and that you may be calling them. Another approach we use is to pull everything we need out of the database. We've managed to work around both the "check and save" and the revision control issues by doing this. We then run the simulations from outside of the design environment.

*Step 3: Getting past the elaboration phase*

A common error/warning during this step are port mismatches. Debugging this is just a matter of going through these messages one by one. In this step, it's especially important to read the error and warning messages carefully. If the problem lies in the model, I suggest fixing the models as you encounter issues, so that they don't linger. But if the design is changing so rapidly that you think changing a particular model will be a futile effort because the design is just going to change again, switch to the schematic view for the block, or auto generate an empty model from the latest schematics and use that if possible.

In the Cadence environment, users often encounter a problem where the module name written in the model does not match the name of the cell. This usually happens as a result of someone copying a cell to another and forgetting to go back to change the module name. The resulting error can often be mysterious, where the elaborator complains of a missing block which seems to be there. This is just something to look for if you see such a message.

Probably the most dreaded error during this phase is the "internal error." This may arguably be the most dreaded of all errors regardless of when it occurs. An internal error is usually the result of the tool seeing something that it didn't expect. It's usually the result of an assertion that has been written into the software that failed. For example, suppose I'm writing software and I expect that a particular input to what I'm writing is always going to be non-zero. Instead of just assuming this and perhaps writing a comment in the code, I write an assertion that checks to see if the input is zero. If the input is zero, I force the program to generate an error and terminate, because I now assume that my code is useless as a key assumption has been violated. I then ask the user to report the problem to me, so that I can figure out if I made an incorrect assumption and have to rewrite my code, or if the problem lies elsewhere. This makes the "internal error" particularly difficult to debug as anything could be the cause.

In the case of the elaborator, it is often the latter case where the internal error is a result of the compiler missing an error and passing along bad compiled code to the elaborator. For example, the compiler may have failed to detect model code that should have been marked illegal. The programmer of the elaborator writes assertions just to double check that the compiler has done its job. However, it was not deemed necessary to spend the time to write comprehensible error messages in the elaborator as these conditions are not expected. In many cases, the elaborator may not have access to what the user originally wrote since there was a compilation phase in between, so any error message it gives even if complete is relatively useless to the user.

The only systematic method I know of to debug this is to take one of two approaches. Start with something that works, maybe empty models for all of the blocks and add the actual models in until something breaks, or start with all of the models with the internal error present and switch models over to empty views until it works. A binary search approach can be taken, where you first add or remove half of the design, then a quarter, and so forth. I like the 'start with something that works' approach. The reason is that empty models run very fast, so I begin with a very fast simulation that will complete without error. These may sound like primitive approaches, but I can usually isolate the problem down to the single line of offending Verilog or Verilog-AMS code in an hour or two, even on a large chip.

*Step 4: Simulating past t=0 in a reasonable amount of time*



Convergence problems are most common at this stage. These are tough to debug. This is why it's critical to follow *Tactic #1*. The next approach is to carefully read the error messages to try to understand what is going on. There's often secondary information in the error messages to assist in debugging that distracts from what is most important to understand. It is critical to track down the primary part of the message.

Common errors at this stage include the analog simulator seeing an "X" or "Z." Loop of shorts is another common error. A loop of shorts is usually where there is more than one ideal voltage source on a node. For these, as with the others, read the error messages carefully to figure out which is the bad node. Ideal switches can often create unexpected consequences including the loop of shorts. Another common message is where prior to the simulator stopping, the simulator reports that it attempted to apply very high voltages to certain nodes. This could be due to current sources that aren't connected. Again, follow the advice in *Tactic #1*. Floating nodes are also a common problem. With models, it's difficult for the simulator to guess where gmin should be added, thus the model writer has to take care to add gmin where necessary.

As a last resort, I go back to the method of replacing models with empty views in the same way I debugged internal error problems to isolate where the convergence issue is being caused.

#### *Step 5: Simulating for a period of time*

A few tasks need to be accomplished in this step. The first is to make sure there are no false positives from the assertions. This is not so much to get rid of the annoyance of reading false positives in the log file, but to ensure that all the blocks that are supposed to turn on do. Recall that assertions that detect errors typically turn off the models. For example, if a power regulator supplying current to some critical analog blocks is off, those analog blocks will never start up and the boot sequence may never complete.

The second is that if the RTL isn't complete, incorrect, or not there at all, some enable signals need to be forced on manually in order for those blocks to be turned on. Here, it is critical to look at the waveforms and match these to the specifications to confirm that your testbench is setting the enable lines to these blocks at the correct time and in the correct sequence. If the RTL is in place, this is where a lack of understanding of the RTL really hurts. Talk to the lead designer or the person who wrote the RTL and re-read the chip-level specifications to try to understand what is going on. If you get blocked and find yourself waiting too long for an answer, go back and start forcing enable signals directly to at least get something started.

The last is to make sure the simulation is running fast. Apply the techniques in *Tactic #2*. Again, as a last resort, I go back to the method of replacing models with empty views to see if I can isolate what is slowing down the simulation.

#### ***Tactic #4: Start as soon as possible***

In every chip-level simulation I've set up or helped to set up, there's one universal constant, and that is we've encountered some issue we've never encountered before. Expected the unexpected! Examples include issues with the revision control system, issues in the foundry provided process design kit, the need to include SystemVerilog when we didn't expect to, discovering that a particular modeling method we used in every model needs to be changed because of an unexpected consequence, and discovering that the people who wrote the digital are no longer around or not available to help. The pattern from this list is that there is no pattern. I just plan that there will be some unexpected issues and that it'll take at least a few days to resolve.

My advice is to not wait until the last minute to work on the chip-level simulations and hope that it will just work. It likely won't! To put this more concretely, even if things aren't completely



ready for a chip-level simulation, push to start the process even if the RTL is only 90% there, the chip-level schematics have barely been drawn, and where only some of the models have been done. Use empty models if necessary. Even consider writing the chip-level netlist yourself.

### ***Tactic #5: Work smart***

You may be tempted to hack through issues just to get the simulation to start. I suggest thinking as you go. Read those error messages. Read and understand the warning messages to see if it really is okay to ignore them. Think about what the RTL is supposed to do. Take half a day and study that system specifications. Walk away from your desk and talk to the digital and analog designers about what you are seeing. Talk to your manager or peers about the problems you are facing, and see if you can talk through some of the problems. They may have seen the issue you're encountering and have a solution at hand. A week can easily go by when employing a trial and error approach to solve a problem. Getting this chip-level simulation started is never a straightforward process. There will be many issues to resolve. You cannot afford to spend a week in a trial and error manner attempting to address a single issue.

### ***Conclusions***

These tactics are not complete, but we hope that these will give you some ideas when preparing for chip-level simulations and debugging chip-level simulation issues. Doing chip-level simulations is a lot of work, and I can only say that despite all of the effort, it has always been worth it from a verification point-of-view. When that chip-level simulation is finally working, I always think of that phrase from the television show, *The A-Team*, "I love it when a plan comes together," an analog verification plan, that is.

Disclaimer: We strive to provide information that is both helpful and accurate. Designer's Guide Consulting, Inc., the creators of this newsletter, makes no representation or guarantees on the newsletter contents and assume no liability in connection with the information contained within it.

Copyright, 2009 © Designer's Guide Consulting, Inc. No reproductions of this newsletter can be made without the express written permission of Designer's Guide Consulting, Inc.

