Digitizing Mixed Signal Verification

Digital Verification Techniques Applied to Mixed Signal and Analog Blocks and System Level Verification

David Brownell and Courtney Schmitt
Analog Devices, Inc.
Norwood, MA
david.brownell@analog.com, courtney.schmitt@analog.com

***Abstract*— Innovation in the field of functional verification has traditionally come from digital design teams, as these teams have led the move from directed test approaches to constrained-random testing, metric-driven environments and industry supported languages and methodologies such as SystemVerilog and the Universal Verification Methodology (UVM) [5]. Analog and mixed signal verification teams have now followed suit, with the introduction of UVM-MS, real number modeling, randomization, and the adoption of traditionally "digital" metric driven verification (MDV) techniques [1,2].**

This paper will discuss how the application of digital verification techniques to analog and mixed signal blocks on a recent design allowed for a more thorough exploration of design robustness, identified bugs potentially missed with traditional analog verification techniques and improved confidence in the taped out design. With the complexity of today's mixed signal designs and the capabilities of mixed signal and analog simulation tools it is no longer sufficient to rely primarily on manual inspection of waveforms for verification, MDV techniques must be applied as they provide the same benefits as they do for digital designs [3].

Keywords—mixed signal, verification, UVM, metric driven, constrained random, co-simulation

I. INTRODUCTION

A. Background of Analog/Mixed Signal DV at ADI

At Analog Devices, design verification (DV) is still a relatively new job title, and only within the last 8 years have fulltime DV engineers been assigned to projects and dedicated DV teams been established across the organization. Before this design engineers typically filled two roles, serving as both the designers of the chip as well as being responsible for the verification.

The lack of dedicated DV engineers has resulted in inconsistent adoption of modern verification methodologies across the company. Within our local organization the processor teams have dedicated DV engineers and have moved fully to a UVM based metric driven verification methodology, while the analog/mixed signal team relies on designers for verification using directed tests and primarily manual inspection of waveforms to determine correctness. The standardization, automation, and visibility that are the foundations of modern digital verification methodologies are missing from the verification methodologies of the analog team.

Our organization recognized that this lack of dedicated verification engineers following a consistent verification methodology for analog and mixed signal designs was a weakness in our development strategy. The LMA project was the first heavily analog product where dedicated DV engineers were allocated to work alongside the analog designers. These engineers had previously focused on DSP SOC based designs and brought with them experience with SystemVerilog, UVM and metric driven verification, but little experience with analog schematic based design and spice simulation. The analog designers were experts in spice, verilog/spice co-simulation, and analog circuit debug but lacked experience with object-oriented programming (OOP) and scripting. This paper documents the results of this collaboration and how the collision of these two worlds resulted in the digitizing of mixed signal verification within our organization.

B. Description of Project

Project LMA represented the development of the LMA processor which is intended to go into systems where up to N LMA chips can be daisy chained to control the transmission/reception of audio data throughout an automobile using an ADI developed protocol. Fig. 1 shows the high level overview of a single LMA processor, which is made up of five analog blocks and one large block of digital control logic.

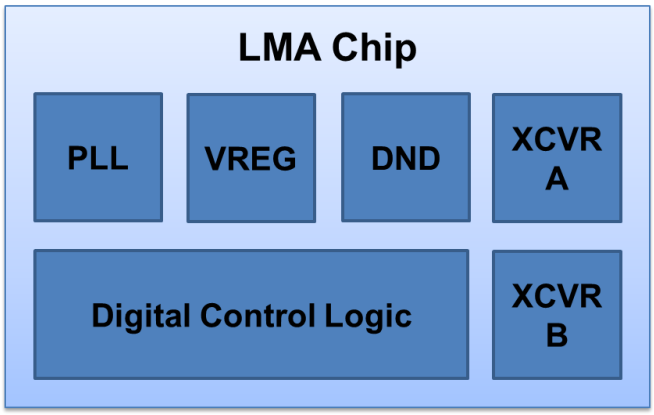

Fig. 1. LMA chip level block diagram

The four major analog blocks are the phase locked loop (PLL), voltage regulator (VREG), downstream node detector (DND), and two instances of a transceiver (XCVR). The PLL is responsible for generating clocks to the digital logic as well as the transceivers. The VREG creates separate supplies for the PLL, digital logic, and DND blocks. The transceiver blocks are responsible for communicating audio data and control data between nodes. XVCR A receives downstream communication while XCVR B handles upstream communication. Finally, the DND block is responsible for determining if downstream nodes exist and are connected properly before the transceivers attempt to communicate. The verification effort for this project included checking that a single LMA chip functioned correctly as well as confirming that any configuration of LMA processors would function as a system.

Fig. 2 shows a high level diagram of a four node LMA system with the slave nodes connected to various audio devices all controlled by a host processor connected to the LMA master node.

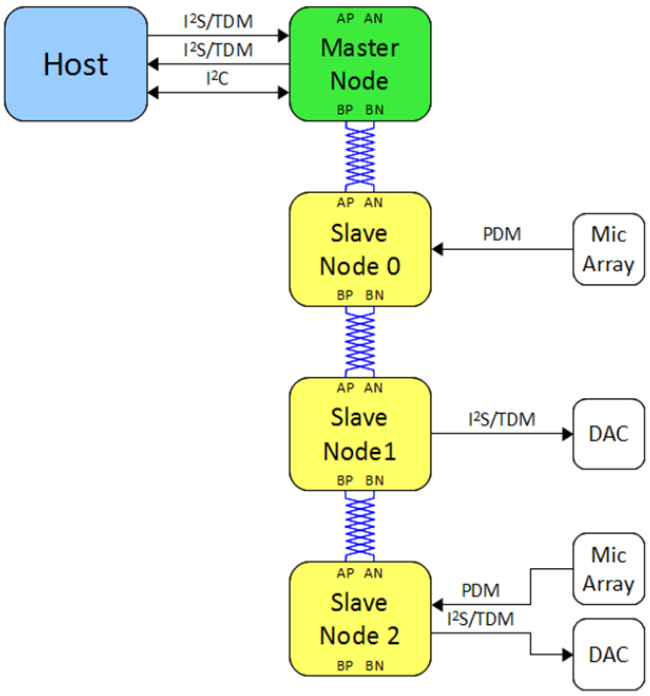

Fig. 2. Four node LMA system diagram

C. *Overview of Digital Techniques Applied to Mixed Signal Verification*

When we began working with the analog designers who were responsible for the verification of these projects in the past, they were able to clearly describe what tests needed to be written, how different variables affected device performance, and what needed to be checked in the design to determine if the devices were operating correctly. What was lacking was any visibility into the process of how they had done this in the past!

Tests were not maintained under revision control and results would be recorded in notebooks to be presented at a design review sometime in the future. There was absolutely no visibility into the verification process or status during the development of a project. For our digital focused projects we follow a standard process as shown in Fig. 3. This process provides consistency and visibility into the verification of a project through the entire development cycle.

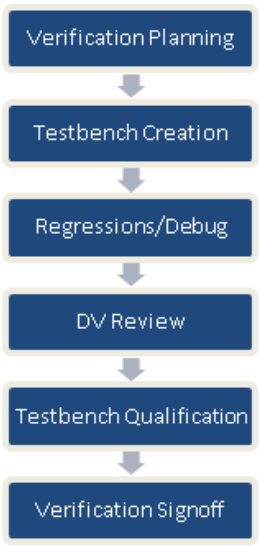

Fig. 3. Digital verification process

Completion of the LMA project required a significant amount of mixed signal and analog verification and coming from a digital verification background we applied the same process as we would have for one of our digital processor based designs. The starting point for the mixed signal verification was the creation of a verification plan that would determine what testbenches needed to be created, what functionality was required in each testbench, and what metrics needed to be tracked to measure the progress of the overall verification effort [3].

Next was the development of block, chip, and system level testbenches. These could be UVM SV based testbenches or schematic based testbenches for analog blocks. In both cases the testbenches were developed to be constrained random and ran self-checking tests, which enabled automated regressions and verification metric collection. Some of the analog and mixed signal blocks were further stressed in system level spice co-simulations to verify the design across multiple LMA nodes.

Finally, DV reviews and functional qualification software were used to look for holes in the verification process and increase the confidence in the overall verification effort.

II. VERIFICATION PLANNING

The biggest surprise when beginning to work on the verification of the analog and mixed signal blocks was the lack of any documentation on what had been done in the past. Unless you could locate the designer who did the work, it was impossible to tell what was done to verify the block.. Testbenches existed, but were not kept under revision control, and were often tweaked on a test by test basis so there was no guarantee that the testbench you looked at was the one used to run a given test. This practice had to be fixed for knowledge sharing and continuous improvement of the block.

The first and one of the most beneficial changes we made was implementing the practice of keeping all testbenches and tests for mixed signal and analog blocks under revision control along with the design schematics. This allows engineers to go back to any point in time and reproduce simulation results on any project. This can be beneficial in terms of learning how to do something or ensuring that the proper verification tasks were done in the first place.

We took a two-step approach to creating the verification plans for LMA. Initially, we got the digital designers, analog designers, and DV engineers together to answer the following questions:

1. What determines if the LMA system is successful and how do we verify that?
2. What blocks have the most impact on LMA system performance and do they need block level testbenches?
3. Since full chip and system level simulations would run forever with full spice models, what can be modeled at a higher level of abstraction and how do we validate the models to ensure we can run legitimate system level simulations?

From this meeting we identified five separate testbenches that needed to be created: PLL block level, VREG block level, DND block level, LMA system level with models for analog blocks, and LMA system level with FastSpice models for most of the analog blocks.

For each of these environments we created a dedicated verification plan by reading the specification and talking to the designers directly to get information not available in the spec. The end result was that for each testbench we had:

1. All the input parameters that needed to be controlled and their legal values
2. All signals that needed to be checked and their correct behavior
3. Required test cases
4. Functional coverage requirements

These verification plans were updated automatically with each regression run and posted to our project DV website. This automated tracking gives managers visibility into the current status of a project at any time. The planning and display also allows for other engineers from across the company to review what is being done for DV and point out areas that need improvement or suggest additional tests. Most important of all when we begin work on the next project that has similar blocks we can re-use the verification plans and testbenches and not have to create them from scratch.

III. SELF-CHECKING TESTS AND REGRESSIONS

The single most effective digital technique applied to the LMA project was simply making the tests in our spice based testbenches self-checking and enabling nightly regressions [1]. Before this project the designers would manually inspect waveforms to determine correctness and while there were a few scripts for some tests to post-process waveforms and perform checks, these also needed to be invoked manually.

Due to the inefficiency of the old manual approach designers would often wait to run tests until a large number of edits had been completed, and even then they may only run a small subset of the full test list as the time to check all the results manually was cumbersome. This often meant that small changes to fix one item would break other features but this might not become known for several weeks until the full set of tests was run again.

The lack of self-checking tests was not due to limitations in the tool suite, there was simply a lack of experience and trust in the practice of automated checking. Without having a designer looking at the waveforms the team was not comfortable believing that the correct behavior was occurring and that real issues could be missed. These same concerns exist in digital verification, but the productivity gains outweigh the risks so additional methods such as coverage and functional qualification have been developed to reduce these risks. We discuss our initial attempts at deploying these techniques later in the paper, and there is a lot of room for innovation in these areas.

For this project we began automating simple checks identified in the verification planning session such as voltage A should never exceed 6.0V, or current B should always be between 600uA and 800uA. We then enabled these checks for all tests and began running nightly regressions. As these checks began to cause tests to fail and debugging the failures identified real circuit issues the designers quickly became more accepting of the automated checking methodology and started asking for new more complicated checks to be written.

With the self-checking tests in place we could then enable nightly regressions which would run all the available tests to ensure that the latest committed design changes did not break previously working functionality or performance requirements.

By default the nightly regression would run all tests, but we also had programmable regression priorities where users could control which tests and how many times each test was run for a given priority. This allowed designers to configure and run short regressions to quickly validate their fixes, or test out various design changes to see which performed best. Then when once they committed there changes the DV team's nightly regression runs with the full test suite would fully validate the changes.

IV. HIERARCHICAL VERIFICATION

The primary outcome of the verification planning sessions was the clear direction for what blocks needed dedicated block level environments, and what could be re-used from the block level environments at the system level.

A. Block Level Environments

1) PLL

The PLL was the first mixed signal block in our team to be verified using a UVM based testbench. Before this, all analog and mixed signal blocks used schematic based testbenches. We chose to do a UVM based environment for the PLL in order to simulate jitter within each LMA node which can have a negative effect on overall system performance. We needed the ability to measure jitter at multiple points across the system and developed a UVM agent specifically for this. Creating this UVM agent allowed us to easily instantiate and control our jitter measurements at the system level. Appendix A, shows the UVM monitor code we created to monitor period jitter, this code can be easily extended to measure CTC and TIE jitter for a signal as well.

2) VREG

Based on some critical safety features implemented in the Voltage Regulator we decided that the VREG should have its own block level TB as well. This testbench was a traditional schematic based design and simulation was done with our in house spice simulator. Changes from previous projects included developing a verification plan for the block with all features to be identified and identifying all tests that needed to be written. Then as these tests were developed and made self-checking we added them to a nightly regression to ensure that design changes to the VREG did not break previously working features.

3) DND

The final block level environment we chose to create was for the DND block which required a Verilog and spice co-simulation. We chose to do a block level verification of the DND for control and performance reasons. This block contained a state machine which required the PLL to be running, and we did not want to verify this block only in the system level TB as the PLL lock sequence could consume a large amount of simulation time for every DND test. With the block level TB it was also easier to control the faults that could cause the downstream nodes to not be detected when compared to the system level environment. Similar to the VREG, the testbench was schematic based and our improvements were to develop a verification plan and create self-checking tests which enabled regression testing and verification tracking.

B. Chip Level TB

The chip level testbench consisted of a UVM environment connected to a single instance of the LMA chip. The testbench environment contained a UVC agent for each of the digital interfaces, as shown in Fig. 4. Each of these agents had a sequencer, driver, and monitor for the interface, in addition to protocol checking assertions and coverage.

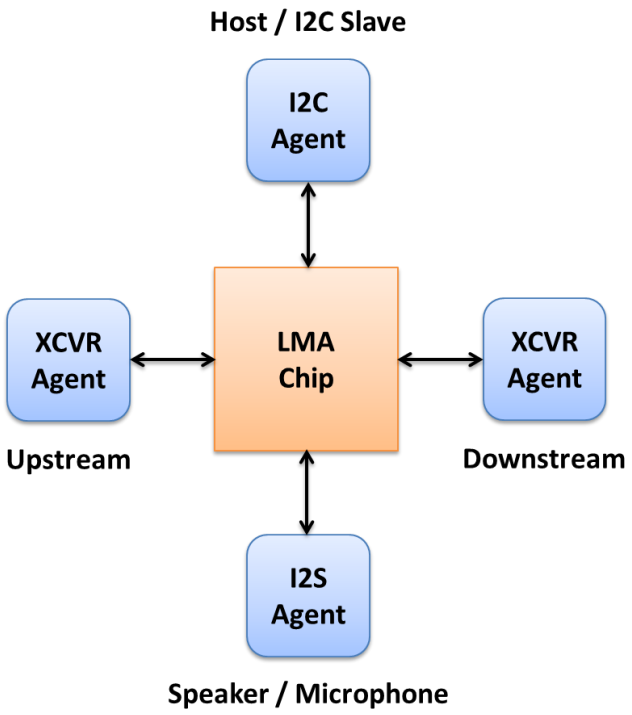

Fig. 4. LMA chip level testbench architecture

As most of the verification effort was focused at the block and system levels, a relatively small amount of time was spent on the chip level testing. As a result, mixed signal blocks were exclusively simulated as RTL functional models in the chip level testbench. The higher-level verification process for the mixed signal blocks is described in detail in the system level testbench description below.

C. System level Testbench Overview

The system level testbench was used to verify the functionality of multiple nodes being daisy-chained together. This was accomplished using a UVM testbench class that created multiple instances of the chip level testbench environment, as shown in Fig. 5. The chip level agents and checkers were automatically included within these chip level environment objects. Additionally, several new scoreboards were created to check the system level interactions between the various nodes.

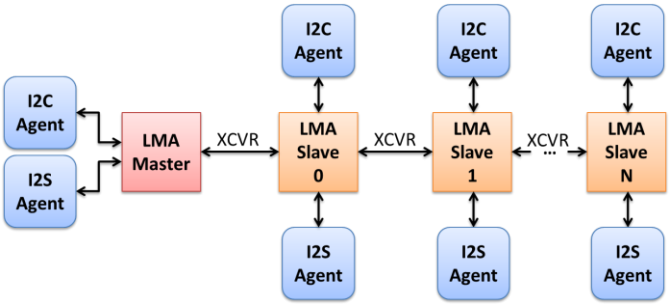

Fig. 5. LMA system level testbench architecture

For analog blocks, models were created at various levels of abstraction in order to optimize mixed signal verification efficiency. For example, system level simulations targeting the verification of digital functionality used RTL functional models of the analog blocks in order to decrease simulation time. Alternatively, system level simulations targeting analog functionality used spice netlists for targeted blocks in order to get the most accuracy. Finally, mixed signal system level simulations used SystemVerilog real number models to get moderate accuracy with a smaller simulation time impact. Fig. 6 shows the various levels of model abstraction used in system level simulations along with their speed and accuracy tradeoff relationships [5,6].

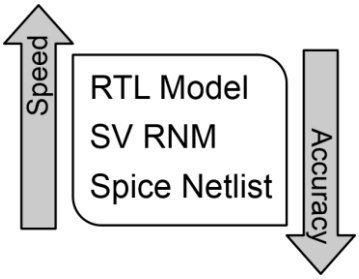

Fig. 6. Mixed signal model abstraction tradeoffs

The selection of which models to use in a simulation was controlled using command line switches to the simulation run script. These command line switches set specific compile defines to include the correct model and any other required compile options. The ease of having a single script option to choose between sets of simulation options was critical in improving the mixed signal verification efficiency.

D. Testbench Re-use

Testbench code re-use was critical to the success of this verification effort. Testbench components were re-used vertically in all levels of the testbench: components developed for the block level were re-used at the chip level, and as previously mentioned, the chip level environment was re-used at the system level. Code re-use relied heavily on the UVM testbench architecture. From the start of the project, verification components were designed with re-use in mind and they were created to be self-contained and configurable so that they could be vertically re-used.

For the mixed signal blocks, the SV RNMs were re-used at all levels of the testbench hierarchy. Since the VREG and DND block level testbenches were primarily spice-based, the amount of re-use was limited. However, since the PLL block level testbench was primarily SystemVerilog, significant re-use of the testbench monitors and checkers at the chip and system levels was possible.

V. SV RNM AND RANDOMIZATION

The LMA product was the first project where we used SystemVerilog and its real number capability to model some of our analog blocks. Real number models combined with the randomization features of SystemVerilog gave us a large boost in productivity based on simulation performance. In the system level simulations each node comes up in series and there is a relatively long wait time for each successive PLL to lock. We could not use spice simulations for these simulations as the runtimes would have been far too long.

Instead, we were able to model the PLL with a SystemVerilog real number model to reduce the lock time of the PLL [6]. In the model we randomized various parameters including temperature variation effects on jitter, skew and voltage settings. We could then randomize each node in the system individually and ensure robust operation against a large number of parameters and values, which would not have been possible with our traditional spice simulation methods. Appendix B, shows a portion of the code we used to randomize jitter for an oscillator on the LMA chip.

With this method we were able to run thousands of simulations and identify potential areas of weakness and then could re-create focused spice level simulations to further debug the issues. Without the real number models to increase efficiency we would not have been able to effectively verify the system level.

In addition to using SV RNM in our digital simulation we also enabled randomization in our spice simulations, which had not been done before in our organization. An example of where this was used was in the DND the testbench schematic which included capacitor values that were required as part of the PCB board design when using an LMA chip. The LMA product specification stated that the capacitor values could be between 1uf and 50uf, so our testbench would randomize the value of each capacitor for every test run to ensure that all combinations would work in the system.

In addition to the capacitor values the testbench would randomize process corner, temperature, input voltage, resistor skew and capacitor skew for each simulation run. Beyond randomizing basic circuit characteristics randomized ramp rates, clock frequencies and the timing of “fault” events that would affect the DND control state machine. An example of a fault would be disconnecting a downstream LMA node in an active system. In this case the DND block is responsible for detecting that the downstream node was removed and driving an output signal to signal the LMA master node to stop communication. The DND control state machine was very complex with a huge number of states and control signals going back and forth to the analog portion of the DND design. We needed to ensure a fault could occur at any time and still be properly detected and found that randomization was the most efficient approach to covering all the scenarios. We could write a single test and use our server farm to run large number of simulations to cover all the cases compared to trying to create directed test cases for all of them.

The analog designers still ran traditional monte carlo and full corner sweep simulations, but found the randomization in our test patterns was beneficial and saved them development time. The setup and run times for traditional monte carlo and full corner sweep simulations were so long that they rarely ran these simulations, only doing so when they had made a large number of design changes. Whereas the randomization in our tests allowed them to quickly run a few cases, cover a range of scenarios, and gain confidence that their updates were not breaking the design.

VI. Functional Coverage in Analog Blocks

Another new technique for us on this project was using functional coverage with analog spice level simulations. As we were using verification plans and defining tests and coverage metrics for the analog mixed signal blocks we also needed the ability to collect functional coverage. Functional coverage is a SystemVerilog construct and was not supported in our spice level simulators.

To handle this we developed a post processing method to enable functional coverage in spice simulations. The simulation was designed to print messages in the spice log file that reported the values of randomized object or voltage and current measurements throughout the simulation. We then post processed this log file with a SystemVerilog program containing the definition of functional coverage groups for these analog constructs and we could collect coverage with this program. This program was automatically run after every simulation and the coverage data was collected and merged into our block level verification plans. A portion of the SV program we used to parse the log files to pull out functional coverage is shown in Appendix C.

Once we had the capability in place we collected two distinct types of functional coverage, which we called "stimulus coverage" and "event coverage". Stimulus coverage was defined for items that were fully controlled by the testbench, such as input voltages, capacitor values, ramp rates, fault type insertion and process skew. Event coverage was defined as observed events in the design or testbench caused by input stimulus. One of the concerns raised when we started using self-checking tests was how do we tell if the checks are actually executed? To alleviate this concern we defined functional "event" covergroups for the checkers in the testbench to ensure they were being executed. Other event coverage items were focused on DUT signals triggering or states of the block control state machines.

These functional coverage groups for analog simulations proved very useful and identified several cases where tests or testbenches were not randomizing values across the full range specified in the verification plan. An example included the capacitor values for the PCB circuit in the DND block level testbench. The specification stated capacitors from 1uf to 50uf were allowed the functional coverage showed that the testbench was only using values from 10uf to 50uf. We updated the testbench to randomize the capacitor values across the full range which caused some of our tests to fail and ultimately required changes in the design. If we had not been using functional coverage with our analog spice simulation we could have missed these items and released a design that did not meet the performance specifications in the datasheet.

VII. System Level Spice Co-simulations

The goal of this section is to provide more detail on the system level mixed signal spice co-simulations that were performed for this project. In the first tapeout of the LMA product, system level co-simulations were not included in the verification strategy. It was believed that the block level analog and mixed signal co-simulations were sufficient for verification of the analog blocks. However, a few issues with the analog blocks were found in the first revision of silicon. These bugs were not found by the block level testbenches because they were caused by interactions between multiple LMA nodes. This prompted an investigation into running system level spice co-simulations before the next scheduled tapeout.

A. Co-simulation Goals and Setup

There were three goals for the system level co-simulations. First, to replicate the issues seen in silicon in the analog blocks. Second, to provide a platform for verification of any design fixes for these issues for the next tapeout. And third, to stress the analog blocks in a system level environment in order to expose any other bugs that might be present in the design. In order for these three goals to be accomplished, the system level simulation environment needed to be highly accurate, but also fast enough to quickly run a large number of simulations. A fast spice simulator, specifically the Cadence Spectre AMSD flow, was selected to meet these speed and accuracy requirements.

The co-simulation environment ran using the same UVM system level testbench described in Section IV.C. All of the digital blocks in the design were compiled in RTL, and specific analog blocks were replaced with spice netlists using the AMSD control files. Additionally, spice models of the components on the application board PCB were included in the co-simulation to accurately model the entire system. Several sets of AMSD control files were created in order to target the different analog blocks by controlling which instances used spice netlists.

B. Replicating Silicon Issues

One of the AMSD configurations focused on verification of the LMA node discovery sequence. This co-simulation used spice netlists for the VREG and DND blocks to ensure that each node in the chain was detected and powered up correctly. This discovery process was of particular interest because there were related issues found in the first revision of silicon which required some software workarounds. After the initial AMSD setup and debug, the co-simulation was able to achieve the first goal of replicating the failing behavior that was observed in silicon. Furthermore, the co-simulation was able to replicate the effects of the software workaround that was being used in silicon, which provided a high degree of confidence in the accuracy of the simulation.

C. Debugging Failures and Testing Design Fixes

Next, the DND block designer was able to use the system level co-simulation results to debug the failing behavior. The ability to debug this issue using a fast simulation that was able to provide accurate voltages and currents was extremely helpful. Once the root cause of the issue was identified, the co-simulation environment was used to test and optimize the design fixes. The co-simulation of the modified design provided confidence that this issue will be fixed in the next silicon revision.

D. Stressing Mixed signal Blocks in the Co-simulation

After this initial success, the third goal of further stressing the mixed signal blocks at the system level was investigated. For the DND block, this meant injecting various fault types as in the block level within the system level co-simulation testbench. This was accomplished by re-using the block level fault model at the system level. The co-simulation AMSD control files were modified to include the spice netlist of the fault model to simulate the various fault conditions during the discovery process. This fault co-simulation was able to expose two new bugs in the design that were not seen at the block level because they were specific to the system level interactions. One of these bugs was only exposed in larger systems (more than five LMA nodes), so would have been impossible to catch without the co-simulation. These bugs prompted further changes to the DND design which were again verified using the system level co-simulation.

E. Co-simulation Testbench Improvements

Once the bugs related to fault injection were resolved, several other improvements were added the co-simulation testbench. First was the ability to run the co-simulation at nominal, fast, and slow process corners. This was accomplished by having three versions of the spice netlists, each one created with a different process skew. The testbench environment was then modified to choose which netlist to run based on a single run script option. This same concept could be applied to running at various temperatures or R/C skew.

The second testbench improvement was to add constrained randomization to the co-simulation. Since the co-simulation re-used the existing UVM system level testbench, the digital portions already had constrained random variables, but the analog portions did not. A PERL script was developed in order to apply constrained random tactics to the analog portions of the co-simulation. This PERL script read in a configuration file, which listed the spice netlist parameters to be randomized in the simulation. This configuration file provided a default value for each parameter as well as minimum and maximum constraints. The PERL script then did a simple randomization of the parameters within the specified constraints and applied them to the spice netlists using the Spectre "alter" command before the start of the simulation. This method was used to randomize several netlist parameters including important resistances and capacitances as well as the input supply voltage at each LMA node.

VIII. Results

A. What worked

The addition of traditionally digital verification techniques to the analog and mixed signal blocks in the LMA project had various benefits that improved both the efficiency and the quality of the DV effort. One of the early benefits for the LMA project came from implementing a formal verification planning process. The creation of a verification plan for each analog block gave a detailed starting point for developing the DV testbenches. Through automated annotation and tracking of the verification plan we identified several issues that would have been potentially missed on previous projects, such as inadequate coverage and missing testcases.

The most significant improvement gained from this work was the development of self-checking tests and regressions. In previous projects, designers were hesitant to change anything in the design because it was difficult to check that they didn't break anything, where now the automated regression system gave analog designers the freedom to explore design changes with a quick turnaround time. The hierarchical verification strategy worked well to test the LMA design at various levels. The development of different models for each analog block was critical to this effort. Accurate spice models enabled detailed block level testing, while the abstraction provided by RTL models allowed for very fast execution of large N-node system level simulations. SV real number models provided a useful combination of accuracy and speed that allowed system level exploration that would not have been possible with standard co-simulation techniques.

Once the self-checking tests for the analog blocks were developed, adding constrained randomization was fairly straightforward. This provided the same benefits that are well known in the digital realm: constrained randomization reduced the number of required directed tests and exercised use-cases that had not been considered. This prompted the addition of functional coverage to the analog testbenches to verify that all input stimulus was applied and required events were observed. The functional coverage collected during regressions provided a good metric to use for verification plan tracking.

The development of a system level Cadence AMSD co-simulation environment was very successful because it provided a high level of accuracy without a large speed tradeoff. This testbench environment enabled the debug of some silicon issues related to LMA node interactions which were not seen at the block-level. The co-simulation enabled design fix exploration and exposed other issues with the silicon that were masked by the original error, which prevented a potential silicon re-spin.

B. What didn't work

During this process there were several notable items that were not successful or were especially challenging to implement. First, we found that it was difficult to develop self-checking tests for the analog spice testbenches. The reason for this was not due to the complexity of coding checks, but rather determining checks that would be valid in all cases. For digital blocks, the designer can usually describe specific scenarios that represent success or failure that can be translated into a check in the testbench. However, in the analog designs this was not so straightforward, as there are so many variables that can slightly modify a voltage or current, such as process skew and temperature. Developing checks that were valid over all corners was an iterative process that required close collaboration with the analog designers.

Another area that provided difficult was model validation. The models we created for the analog blocks greatly increased simulation speed and overall efficiency, but caused some issues due to model inaccuracies. For one of the blocks, some items were incorrectly modeled and were not identified before

tapeout, resulting in silicon issues. The model validation that was performed on this project was limited to some spice comparisons and peer review. In the future, we need to develop an automated model validation system to prevent inaccuracies.

A final difficultly identified in this effort during was the ability to run high-accuracy simulations at fast speeds. This concern is not limited to the LMA project, but was very important due to the number of analog blocks and the importance of system level interactions in our multi node simulations. The AMSD spice and SV co-simulation environment helped bridge the gap between speed and accuracy, but still had limitations that hindered regression throughput. The tradeoffs and restrictions imposed by simulation tools required careful planning in order to maximize DV efficiency at all simulation levels.

C. Metrics

The goal of integrating digital verification techniques into our analog and mixed signal verification process was to improve efficiency and ultimately find more bugs than we were capable of with our previous methods. Exact comparisons to previous projects are not possible as before this project the mixed signal verification effort did not include tracking of bugs found, regression history or simulation performance.

However, there is no doubt that by using self-checking tests, automated regressions and randomization we were able to run a much larger number of simulations for the LMA project than on previous projects. This was especially true at the system level where SV RNM allowed us to exercise many thousands of scenarios which would not have been possible in the past. The number of bugs found, simulations run counts and model performance comparison metrics are shown in the various tables below.

1) Bugs Found

Testbench	Bugs Found
PLL Block Level	16
VREG Block Level	18
DND Block Level	11
System Level	238

2) Simulation Counts

Testbench	Sim Count
PLL Block Level	2463580
VREG Block Level	249477
DND Block Level	810281
System Level	329702

3) Simulation and Model Performance

a) PLL Block Level Testbench

Simulation Type	Runtime
Verilog & SV RNM	5 Minutes
Verilog + fast spice Cosim	24 Hours
Verilog & Spice Cosim	1 Week

b) VREG & DND Block Level Testbenches

Simulation Type	Runtime
VREG - Spice Only	5-10 Minutes
DND - Verilog & Spice Cosim	1-3 Minutes

c) System Level Testbench

Simulation Type	TB Cfg	Runtime
Spice Only	1-9	Not Possible
AMSD Cosim (Spice for DND, PLL, VREG)	2 Node	6 Days
	9 Node	Not Possible
AMSD Cosim (Verilog PLL, Spice VREG & DND)	2 Node	13 Minutes
	9 Node	1.25 Hours
Verilog Sim (SV RNM for PLL, VREG & DND)	2 Node	9 Minutes
	9 Node	1.5 Hours
All Verilog Models	2 Node	30 seconds
	9 Node	2.5 Minutes

IX. Conclusion

While it was not always an easy road, the application of digital verification techniques to mixed signal and analog blocks proved very useful to us on the LMA project. The hallmark of digital verification techniques is to automate as much as possible and this project demonstrated that mixed signal and analog verification can and should take advantage of the power of automation. The positive results achieved during the LMA project have shown that the methods described in this paper represented a significant improvement over our existing verification strategy for analog and mixed signal blocks. As a result, we plan to continue applying digital verification techniques to analog and mixed signal blocks in all of our future products at ADI.

Acknowledgments

We would like to thank Stuart Patterson, Ara Arakelian, Lew Lahr and William Hooper for their help and support on the LMA project.

References

[1] N. Khan, Y. Kashai, and H. Fang, "Metric driven verification of mixed-signal designs," DVCON 2011.

[2] N. Khan and Y. Kashai, "From spec to verification closure: a case study of applying UVM-MS for first pass success to a complex mixed-signal SoC design," DVCON 2012.

[3] N. Khan, G. Glennon, and D. Romaine, "MS-SoC best practices – advanced modeling & verification techniques for first-pass success," DVCON 2013.

[4] S. Balasubramanian and P. Hardee, "Solutions for mixed-signal SoC verification using real number models," Cadence white paper, http://www.cadence.com/rl/Resources/white_papers/Mixed_Signal_Verification_wp.pdf, 2013.

[5] A. Milne and D. Roberts, "Utilizing digital techniques for analog and mixed-signal verification," Synopsys white paper, http://www.synopsys.com/tools/verification/amsverification/capsulemodule/customsim-utidigitaltech-wp.pdf, 2010.

[6] F. Strumble, "Advanced verification techniques for the mixed-signal domain," CDNLive EMEA 2012.

Appendix A

```
class clk_monitor extends uvm_monitor;

 real first_period; //First period identified by monitor
 real start_time;   //When clk_monitor will begin to collect information
 real stop_time;    //stop time if needed
 real ideal_period; //If defined then this period is used for calculations

 //Period jitter specs
 real allowed_period_jitter;
 real allowed_max_duty_cycle;
 real allowed_min_duty_cycle;

 //Period Jitter values
 real this_period;
 real period_jitter;
 real max_period;
 real min_period;

 //Duty cycle values
 real max_duty_cycle;
 real min_duty_cycle;

 string name;
 virtual interface clk_monitor_interface vif;

 // component macro
 `uvm_component_utils_begin(clk_monitor)
 `uvm_component_utils_end

 // component constr - required syntax for UVM automation and utilities
 function new (string name, uvm_component parent);
  super.new(name, parent);
  this.name = name;
  //Default check values
  start_time = 1;
  allowed_max_duty_cycle = 0.55;
  allowed_min_duty_cycle = 0.45;
  allowed_period_jitter  = 1000;
 endfunction : new

 function void build_phase(uvm_phase phase);
  super.build_phase(phase);
 endfunction: build_phase

 // start_of_simulation
 function void start_of_simulation_phase(uvm_phase phase);
  super.start_of_simulation();
  `uvm_info(get_type_name(), {"start of simulation for ", get_full_name()},
UVM_HIGH)
 endfunction : start_of_simulation_phase

 // UVM run() phase
 task run_phase(uvm_phase phase);

  real first_posedge, first_negedge;
  `uvm_info(get_type_name(), "Inside the run() phase", UVM_MEDIUM)
  void'(uvm_config_db#(real)::get(this,"","start_time",start_time));
  `uvm_info(get_type_name(), $psprintf("start_time set to %d", start_time),
UVM_MEDIUM)
   #(start_time);

  // Check that an interface was connected
  if(!uvm_config_db#(virtual clk_monitor_interface)::get(this,"","vif",vif))
   `uvm_fatal("NOVIF", {"virtual interface must be set for: ",
get_full_name(),".vif"});

    wait (mask_bit == 0);

  `uvm_info(get_type_name(), "Starting Jitter measurement",
UVM_MEDIUM)

   @(posedge vif.clk);
   first_posedge = $realtime;

   @(negedge vif.clk);
   first_negedge = $realtime;

   @(posedge vif.clk);
   first_period = $realtime - first_posedge;
   max_duty_cycle = (first_negedge-first_posedge)/first_period;
   min_duty_cycle = (first_negedge-first_posedge)/first_period;

   duty_cycle_checks();
   `uvm_info(get_type_name(), $psprintf("start %0.4f, first_pos %0.4f,
first_neg %0.4f, first_period %0.4f!",start_time, first_posedge, first_negedge,
first_period), UVM_MEDIUM)
   `uvm_info(get_type_name(), $psprintf("First period %0.4f!",first_period),
UVM_MEDIUM)

   //If no ideal period defined then first period is used as ideal.
   if (ideal_period == 0.0)
    begin
     ideal_period = first_period;
    end

  `uvm_info(get_type_name(), $psprintf("Ideal Period %0.4f!",ideal_period),
UVM_MEDIUM)
```

```
    //Once initial period defined kick off jitter monitors
    fork
      monitor_period_jitter();
      monitor_duty_cycle();
    join
  endtask : run_phase

  // UVM report_phase
  function void report_phase(uvm_phase phase);
    `uvm_info(get_type_name(), $sformatf("Report: CLK Jitter Monitor Done"),
UVM_MEDIUM)

    `uvm_info(get_type_name(), $psprintf("Max Period = %3.4f ps",
max_period), UVM_MEDIUM)
    `uvm_info(get_type_name(), $psprintf("Min Period = %3.4f ps",
min_period), UVM_MEDIUM)

    if ((max_period ==0) || (min_period == 0)) `uvm_error(get_type_name(),
$psprintf("Never observed clk toggling!"))

    `uvm_info(get_type_name(), $psprintf("Max Duty Cycle = %2.4f",
max_duty_cycle), UVM_MEDIUM)
    `uvm_info(get_type_name(), $psprintf("Min Duty Cycle = %2.4f",
min_duty_cycle), UVM_MEDIUM)

  endfunction : report_phase

  extern task monitor_period_jitter();
  extern task period_jitter_checks(bit max_check);
  extern task monitor_duty_cycle();
  extern task duty_cycle_checks();
endclass : clk_monitor

////////////////////////////////////////////////////////////////////////
task clk_monitor::monitor_period_jitter;

  real last_edge, this_edge;
  last_edge = $realtime;

  max_period = ideal_period;
  min_period = ideal_period;

  forever
   begin
    @(posedge vif.clk);
    this_edge = $realtime;
    this_period = this_edge - last_edge;

    if (!mask_bit) begin

    //Measure period_jitter
    if (this_period > max_period)
     begin
       max_period = this_period;
     `uvm_info(get_type_name(), $psprintf("New Max Period = %3.4f ps",
max_period), UVM_HIGH)
      period_jitter_checks(1);
     end
    if (this_period < min_period)
     begin
               min_period = this_period;
     `uvm_info(get_type_name(), $psprintf("New Min Period = %3.4f ps",
min_period), UVM_HIGH)
      period_jitter_checks(0);
     end
    end

    //Prepare for next cycle
    last_edge   = this_edge;

  end
endtask

////////////////////////////////////////////////////////////////////////
task clk_monitor::period_jitter_checks(bit max_check);

  if (!mask_bit) begin
  //Period jitter check 1
  if (max_check && ((max_period - ideal_period) > allowed_period_jitter))
   begin
    period_jitter = max_period - ideal_period;
   `uvm_error(get_type_name(), $psprintf("Observed Pos Period Jitter of
%3.4f ps exceeds maximum allowed of %3.4f ps in
%s",period_jitter,allowed_period_jitter, name));
   end

  //Period jitter check 2
  if (!max_check && ((ideal_period - min_period) > allowed_period_jitter))
   begin
    period_jitter = ideal_period - min_period;
   `uvm_error(get_type_name(), $psprintf("Observed Neg Period Jitter of
%3.4f ps exceeds maximum allowed of %3.4f ps in
%s",period_jitter,allowed_period_jitter, name));
   end
  end

endtask

////////////////////////////////////////////////////////////////////////
task clk_monitor::monitor_duty_cycle();

  real this_dc_period;
  real last_posedge, last_negedge;
  real this_duty_cycle;

  last_posedge = $realtime;

  forever
   begin
    @(negedge vif.clk);
    last_negedge = $realtime;

    @(posedge vif.clk);
    this_dc_period = $realtime - last_posedge;
    this_duty_cycle = ($realtime - last_negedge)/this_dc_period;
    last_posedge = $realtime;

    if (!mask_bit) begin
     if (this_duty_cycle > max_duty_cycle)
      begin
               max_duty_cycle = this_duty_cycle;
     `uvm_info(get_type_name(), $psprintf("New Max duty cycle = %2.2f",
max_duty_cycle), UVM_HIGH)
      duty_cycle_checks();
     end
    if (this_duty_cycle < min_duty_cycle)
     begin
               min_duty_cycle = this_duty_cycle;
     `uvm_info(get_type_name(), $psprintf("New Min duty cycle = %2.2f",
min_duty_cycle), UVM_HIGH)
      duty_cycle_checks();
     end
    end
  end
endtask
```

```
////////////////////////////////////////////////////////////////////////
task clk_monitor::duty_cycle_checks;

  if (!mask_bit) begin

  //Max duty cycle check
  if (max_duty_cycle > allowed_max_duty_cycle)
   `uvm_error(get_type_name(), $psprintf("Observed Duty Cycle %2.4f
exceeded maximum allowed of
%2.4f",max_duty_cycle,allowed_max_duty_cycle));

  //min duty cycle check
  if (min_duty_cycle < allowed_min_duty_cycle)
   `uvm_error(get_type_name(), $psprintf("Observed Duty Cycle %2.4f below
minimum allowed of %2.4f",min_duty_cycle,allowed_min_duty_cycle));

  end
endtask
```

APPENDIX B

```
module
pll_bias_osc(osccclk,porb,ibias2p5,PLLVDD,PLLGND,DVDD,DGND,hys,osc_pc,o
sc_tc);
  input       porb;
  input       hys;
  input [7:0]   osc_pc;
  input [6:0]   osc_tc;
  inout       PLLGND;
  inout       PLLVDD;
  inout       DGND;
  inout       DVDD;
  output       osccclk;
  output       ibias2p5;

`define OSCCLK_FREQUENCY 31.00
`define OSCCLK_DUTY 0.5

  int         osccclk_freq_int;
  real        osccclk_freq;
  real        osccclk_duty;
  real        osccclk_h;
  real        osccclk_l;

  int         jitter_int;
  real        jitter;
  real        jitter_acc;
  int         fh_jitter;//debug

  real        drift;
  real        wander_rate;    //period wander of pll in ps per second.
  int         wander_freq_int;
  real        wander_freq;
  int         curr_wander_freq_int;

  reg         osccclki;
  reg         osccclk_int;
  reg [8:0]    osccclk_cnt;
  reg         porb_delayed;

  //Testbench control
  bit         allow_pll_bias_osc_jitter;
  bit         allow_pll_bias_osc_wander;
  int         wander_start_time;
  int         wander_adjust_rate;
  int         osc_seed;
  int         osccclk_duty_cycle;
  int         osccclk_wander_rates[25];
  int         osccclk_wander_rates_index;
  bit         apply_jitter;
  int         osc_stdev_jitter;

  initial begin
    jitter_acc = 0.0;
    osccclk_freq = 31.00;
    wander_freq = osccclk_freq;
    osccclk_duty = 0.5;
    apply_jitter = 0;
    jitter     = 0.0;
    osccclk_h = osccclk_duty*1e6/osccclk_freq;
    osccclk_l = (1-osccclk_duty)*1e6/osccclk_freq;
    porb_delayed = 0;

    #3;

      `uvm_info("OSC_MODEL", $sformatf("Allow pll_bia_osc jitter = %b",
allow_pll_bias_osc_jitter), UVM_MEDIUM);
      `uvm_info("OSC_MODEL", $sformatf("Allow pll_bia_osc wander = %b",
allow_pll_bias_osc_wander), UVM_MEDIUM);
      `uvm_info("OSC_MODEL", $psprintf(" %m Seed = %d", osc_seed),
UVM_MEDIUM);

    osc_seed       = $urandom(osc_seed);
    //osccclk_freq_int = $urandom_range(3255,2945);   //in Mhz, Osc freq
should be 31Mhz, This is +/- 5%
    osccclk_freq     = osccclk_freq_int/100.0;

    //Randomize duty cycle if either wander or jitter is allowed
    if ((allow_pll_bias_osc_jitter) || (allow_pll_bias_osc_wander)) begin
      osccclk_duty = osccclk_duty_cycle/100.0;      //Duty cycles 45%-55%/
    end

    // Print osc_stdev_jitter (0 if not allowed)
    if (!allow_pll_bias_osc_jitter) begin
      osc_stdev_jitter = 0;
    end

    `uvm_info("OSC_MODEL", $sformatf("Oscillator Jitter Standard Deviation
= %d pS", osc_stdev_jitter), UVM_MEDIUM);

    //Wander of oscillator limited to 50Khz per second
    if (allow_pll_bias_osc_wander) begin
      wander_freq_int = osccclk_freq_int +
osccclk_wander_rates[osccclk_wander_rates_index++];
      wander_freq     = wander_freq_int/100.0; //Wander freq +/- 50Khz of
oscfreq
    end

      `uvm_info("OSC_MODEL", $sformatf("osc clk freq         = %0.2f Mhz",
osccclk_freq), UVM_MEDIUM);
      `uvm_info("OSC_MODEL", $sformatf("osc clk period       = %0.2f ns",
(1e3/osccclk_freq)), UVM_MEDIUM);
      `uvm_info("OSC_MODEL", $sformatf("osc clk duty         = %0.2f ",
osccclk_duty), UVM_MEDIUM);


    //Calculate osclk high and low pulse widths based on freq and duty cyle
    osccclki = 0;
    osccclk_h = $floor(osccclk_duty*1e6/osccclk_freq);
    osccclk_l = $floor((1-osccclk_duty)*1e6/osccclk_freq);

  end
```

```
  ////////////////////////////////////////////////
  //Osc Wander control
  ////////////////////////////////////////////////
  initial begin
    #3;

    if (allow_pll_bias_osc_wander) begin
      //Start wander at time in cfg specified in ms so converted to ps
      #(wander_start_time*1e9);

      //Determine wander rate in ps and associated drift;
      //Drift based on wander rate in ns divided by number of clock periods in
1s
      wander_rate = $floor(((1e6)/oscclk_freq)-((1e6)/wander_freq));
      drift = wander_rate / (oscclk_freq*1e6);

      `uvm_info("OSC_MODEL", $sformatf("%m osc wander tgt freq   = %0.2f
Mhz", wander_freq), UVM_MEDIUM);
      `uvm_info("OSC_MODEL", $sformatf("%m osc wander tgt period =
%0.2f ns", (1e3/wander_freq)), UVM_MEDIUM);
      `uvm_info("OSC_MODEL", $sformatf("%m osc clk wander rate   = %0.4f
ps per second", wander_rate), UVM_MEDIUM);
      `uvm_info("OSC_MODEL", $sformatf("%m osc clk drift         = %0.12f per
clock edge", drift), UVM_MEDIUM);

      if (wander_adjust_rate > 0) begin
        while (1) begin
          //Move to next adjust time
          #(wander_adjust_rate*1e9);

          //Adjust target  based on current oscclk freq/period which depends
on how much drift has taken place
          curr_wander_freq_int = $floor((1.0/(oscclk_h + oscclk_l))*1e8);
          wander_freq_int = curr_wander_freq_int +
oscclk_wander_rates[oscclk_wander_rates_index++];

          if (wander_freq_int > 3235) wander_freq_int = 3235; //Do not
exceed boundaries
          if (wander_freq_int < 3045) wander_freq_int = 3045; //Do not
exceed boundaries

          //New drift calculation based on current oscclk_h and oscclk_l
          curr_wander_freq_int = $floor((1.0/(oscclk_h + oscclk_l))*1e8);

          wander_freq     = wander_freq_int/100.0; //Wander freq +/- 50Khz
of oscfreq
          wander_rate = $floor(((1e6)/(curr_wander_freq_int/100.0))-
((1e6)/wander_freq));
          drift = wander_rate / ((curr_wander_freq_int/100.0)*1e6);

          `uvm_info("OSC_MODEL", $sformatf("%m New osc wander tgt freq
= %0.2f Mhz", wander_freq), UVM_MEDIUM);
          `uvm_info("OSC_MODEL", $sformatf("%m New osc wander tgt
period = %0.2f ns", (1e3/wander_freq)), UVM_MEDIUM);
          `uvm_info("OSC_MODEL", $sformatf("%m New osc clk wander rate
= %0.4f ps per second", wander_rate), UVM_MEDIUM);
          `uvm_info("OSC_MODEL", $sformatf("%m New osc clk drift       =
%0.12f per clock edge", drift), UVM_MEDIUM);
        end
      end
    end
  end


  //oscclk will not be driven for at least 500ns after porb is released
  always @(posedge porb)
   begin
    #500ns;
    if (porb === 1'b1) porb_delayed = 1'b1;
   end

  always @(negedge porb)
   begin
    porb_delayed = 1'b0 ;
   end

  //Clock generation with jitter
  always begin
    #(oscclk_l) begin
      //oscclki = porb;
      oscclki = porb_delayed;
    end
    jitter_calc();
    #(oscclk_h+jitter) oscclki = 0;
    oscclk_h = oscclk_h + drift ;
    oscclk_l = oscclk_l + drift ;
  end
  assign ibias2p5 = porb;

  always @(posedge oscclki or negedge porb) begin
    if (~porb) begin
      oscclk_cnt[8:0] <= 9'h000;
    end
    else begin
      if (oscclk_cnt[8:0] < 9'h1ff)
        oscclk_cnt[8:0] <= oscclk_cnt[8:0] + 1'b1;
    end
  end

  always @(oscclki or porb) begin
    if (~porb) oscclk_int <= 1'b1;
    else begin
      if (oscclk_cnt[8:0] >= 9'h020)
        oscclk_int <= 1'b0;
      if (oscclk_cnt[8:0] >= 9'h02f)
        oscclk_int <= oscclki;
    end

    if (~porb) apply_jitter <= 1'b0;    // allows monitor to measure one pefect
period
    else if (oscclk_cnt[8:0] >= 9'h030)
      apply_jitter <= 1;
  end

  task jitter_calc;
    if (allow_pll_bias_osc_jitter) begin
      if (apply_jitter) begin
        jitter_int = $dist_normal(osc_seed, 0, osc_stdev_jitter);
        jitter = 1.0*(jitter_int);
        jitter_acc = jitter_acc + jitter;
      end
      else begin
        jitter = 0;
      end
    end
  endtask

  assign oscclk = oscclk_int;

endmodule
```

APPENDIX C

```
program test;

typedef enum {NO_SKEW, FAST, TYP, SLOW} skew_t;
typedef enum {NO_TEMP, COLD, NOM, HOT}  temp_t;

class tb_cov;

  int fh, file_status;
  string line;

  skew_t skew;
  temp_t temp;
  int VOUT1_cap;

  covergroup cov_cg;

    skew_cp : coverpoint skew { ignore_bins ign_bins = {NO_SKEW}; }
    temp_cp : coverpoint temp      { ignore_bins ign_bins = {NO_TEMP}; }

    vout1_cap_cp : coverpoint VOUT1_cap {
      bins VOUT1_cap_vals [] = {[1:50]};
    }
  endgroup

  ////////////////////////////////////////////////
  virtual function void get_cov();
   string temp_str;
   string testname;

   fh = $fopen(testname, "r");
   file_status = $fgets(line, fh);
   while (file_status)
    begin
     if (line[0] == "N")
      begin

      //Check for SKEW
       if (str_match(line, "SKEW"))
        begin
         if      (line[8] == "F") skew = FAST;
         else if (line[8] == "S") skew = SLOW;
         else if (line[8] == "N") skew = TYP;
        end

       //Check for TEMP
       if (str_match(line, "TEMP"))
        begin
         if      (line[8] == "-") begin temp = COLD; $display("COLD"); end
         else if (line[8] == "1") begin temp = HOT; $display("HOT"); end
         else if (line[9] == "2") begin temp = NOM; $display("NOM"); end
         $display("%s", line);
        end

       //Check for VOUT1 Cap value
       if (str_match(line, "VOUT1"))
        begin
          temp_str = line.substr(21,24);
          if (line[line.len()-2] == "5") VOUT1_cap = $floor(temp_str.atoreal*10);
          if (line[line.len()-2] == "6") VOUT1_cap = $floor(temp_str.atoreal);
          if (line[line.len()-2] == "7") VOUT1_cap = $floor(temp_str.atoreal/10);
          $display("%s %s %d", line, temp_str, VOUT1_cap);
        End

     file_status = $fgets(line, fh);
    end
   $fclose(fh);

    //Update coverage
    vreg_cg.sample();

   endfunction: get_cov

   function new();
     vreg_cg = new();
   endfunction

endclass: tb_cov

////////////////////////////////
tb_cov tb_cov1;

initial
begin

  tb_cov1 = new();
  tb_cov1.get_cov();

end

endprogram: test
```