

Using Digital Verification Techniques on Mixed-Signal SoCs with CustomSim and VCS

March 2011

Authors **Abstract**

Graeme Nunn
Calvatec

Fabien Delguste
Adiel Khan
Abhisek Verma
Bradley Geden
Synopsys

The traditional approach used for verification in the analog world still lacks some key aspects that have been efficiently deployed in digital verification for years. SPICE-based analog verification environments are usually hard to reuse at the system-on-chip (SoC) level, difficult to control and barely meet the required simulation performance. By leveraging the well-proven VMM and UVM methodologies, the VCS[®] AMS testbench technology is to provide analog designers and verification engineers with a methodology that allows them to:

- ▶ Introduce analog verification planning
- ▶ Introduce constraint-random verification for driving analog nodes
- ▶ Model analog stimulus as shaped transaction-based bus functional models
- ▶ Integrate reference models with various abstraction level
- ▶ Sample analog nodes to monitor incoming traffic
- ▶ Introduce assertions on analog nodes
- ▶ Introduce analog code coverage and functional coverage
- ▶ Introduce regression management

In addition to elaborating on the above features, this white paper describes a scalable and reusable methodology for verifying analog IP. Reuse is made possible by correct modeling of verification models that can be stitched into the SoC. These models can be implemented with HDL or Verilog-AMS, depending on the required accuracy.

This white paper is a case study that explains the various aspects of this methodology that can be applied to VMM/UVM, from verification planning to testbench implementation and coverage collection.

Introduction

Next-generation SoCs contain an increasing number of different analog IPs. For example, it is becoming the norm to see multiple standard interfaces, such as USB, Ethernet, SATA, DDR, etc., on a single chip. Additionally, there is a need for more analog IP to handle multiple power domains, clock generation (PLL) and conversion (ADC, DAC).

As the need to include more embedded analog IP increases, it is challenging to architect a verification environment that can accommodate both digital and analog verification. The VCS AMS testbench technology provides a solution that helps to fill this gap. As shown in Figure 1, the technology complements a traditional digital verification environment with a few components that can drive some analog IPs, such as ADCs or clock generation.

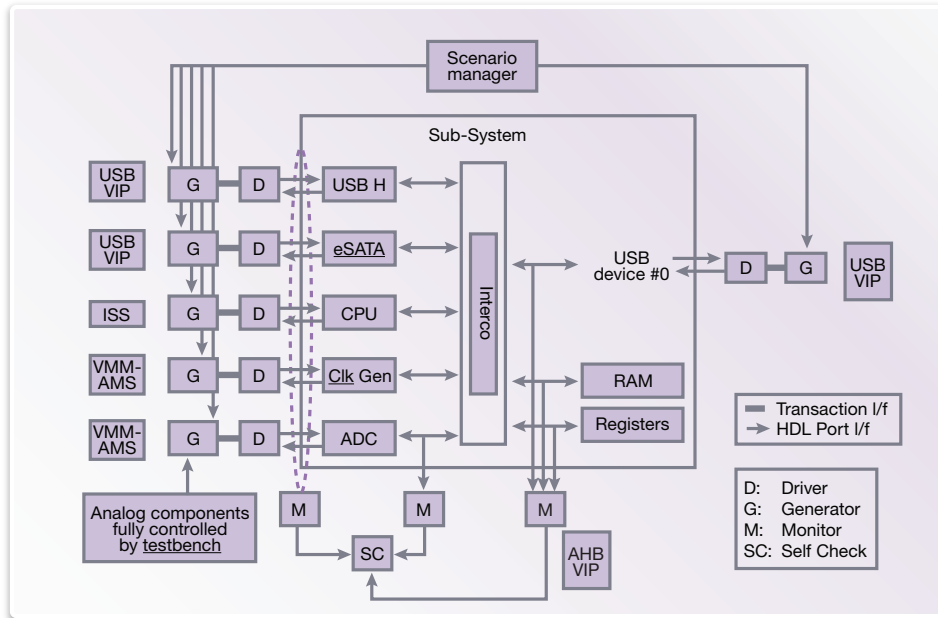


Figure 1: Verifying Digital and Analog Blocks

With this architecture, it is possible to decide when to start injecting analog traffic, when to stop, and when to sample the output results. For example, consider a typical scenario to initialize and configure the subsystem registers, wait for the clock generation to stabilize, start injecting analog ramps to the ADC and read the internal converted digital output whenever the SoC receives an interrupt.

Certainly, this architecture allows you to use other VMM or UVM base classes and applications, such as RAL, to initiate register traffic and VMM-LP to model the low-power domains. Based on the desired accuracy, the ADC can either be a transistor-level SPICE netlist or a reference model. The latter provides faster simulation performance but with less accuracy. This reference model can either be written in Verilog-AMS or SystemVerilog®.

As shown in Figure 2, the VCS AMS testbench technology comes with built-in base classes that allow driving analog inputs with given shapes, such as sine, sawtooth, square and white noise. These source generators can be combined to create specific shapes, for example, to add a sine waveform with a given maximum/minimum voltage and frequency with well-distributed noise.

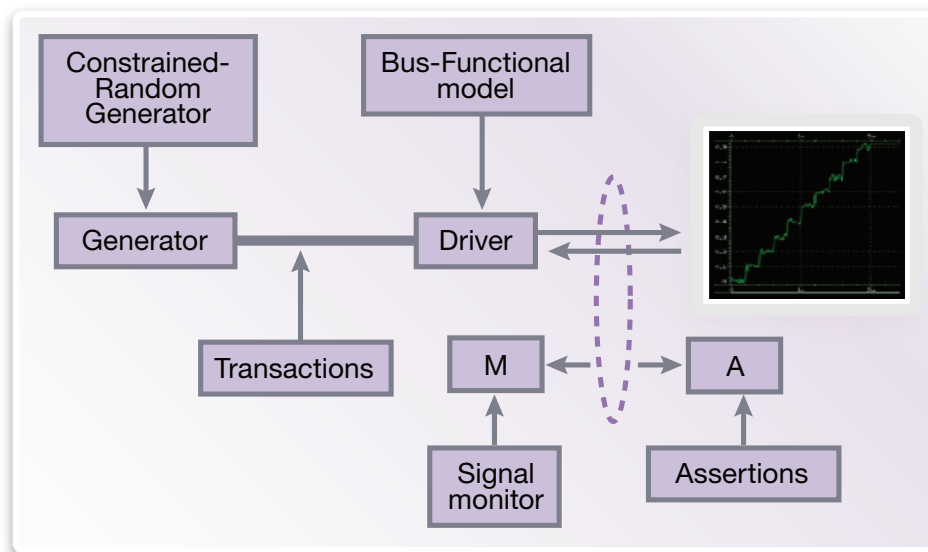


Figure 2: VCS-AMS Components

You can also use these base classes to model your own traffic shapes. An interesting application is to directly inject voltage waveforms anywhere in your analog IP. This is of particular interest for pipelined IPs or staged designs where you can skip the first stage and directly inject a given waveform to the second stage input. This approach allows you to speed up the simulation time without having to wait for the first stage to be ready. However, as this waveform is modeled in SystemVerilog, you can model it in a few lines of code with specific traffic shapes, which is difficult to achieve in SPICE.

Architecture

The VCS AMS testbench technology enables an engineer to easily connect a top SystemVerilog environment with an analog netlist, which can be either in Verilog-AMS or SPICE. A very important aspect of this methodology is to enable the possibility to drive and sample an analog node directly from a SystemVerilog component or module. To achieve this, VCS comes with direct communication between a SystemVerilog real and an analog node.

With this communication, it becomes possible to have fine grain resolution to:

- ▶ Drive a SPICE electrical node by connecting it to a SystemVerilog real
- ▶ Sample a SPICE electrical node by converting it to a SystemVerilog real

Similarly, it is also possible to convert electrical to logic, i.e.,

- ▶ Drive a SPICE electrical node by connecting it to SystemVerilog logic
- ▶ Sample a SPICE electrical node by converting it to SystemVerilog logic

It is important to understand that the overall communication between SystemVerilog and SPICE is done with real. Therefore, all analog DUT nodes can be grouped in a single SystemVerilog interface. So, as in digital verification, SystemVerilog modports and clocking blocks can be introduced to determine analog node directions (input, output, in/out) and synchronization against a reference node, such as a clock.

As stated in the assertion section, it also becomes possible to gather analog assertions in this interface. This approach allows modeling interfaces efficiently that are easily reusable between projects and higher levels of integration (i.e., from IP level to SoC level). Now that we've solved the interfacing between SPICE and SystemVerilog, all verification techniques that are standard practice in digital verification can be fully leveraged. For instance, the verification environment can be architected with:

- ▶ Interfaces for electrical-to-real or real-to-electrical
- ▶ Generators to drive analog inputs
- ▶ Samplers to strobe analog output; these samplers can be combined with scoreboard or reference models to ensure the SPICE DUT is behaving as expected
- ▶ Configurations, which can be shared with the SPICE DUT
- ▶ Assertions that can be on the SPICE boundary nodes or internal

However, it is also possible to cover analog nodes. For instance, if an analog node is known to swing between 0.4V to 0.8V, it is fairly simple to associate this variation to toggle coverage and ensure the signals have moved as expected. Figure 3 shows a typical architecture that can be used to verify an analog IP with a VCS AMS testbench.

Extra attention is required during the self-checking capability of this testbench. If the analog block is simply a transfer function, i.e., $y = f(x)$, then the reference model can be written in SystemVerilog with the help of real scalars to model x , y and the transfer function. If the analog block is more complicated, then it makes sense to use a language such as Verilog-AMS.

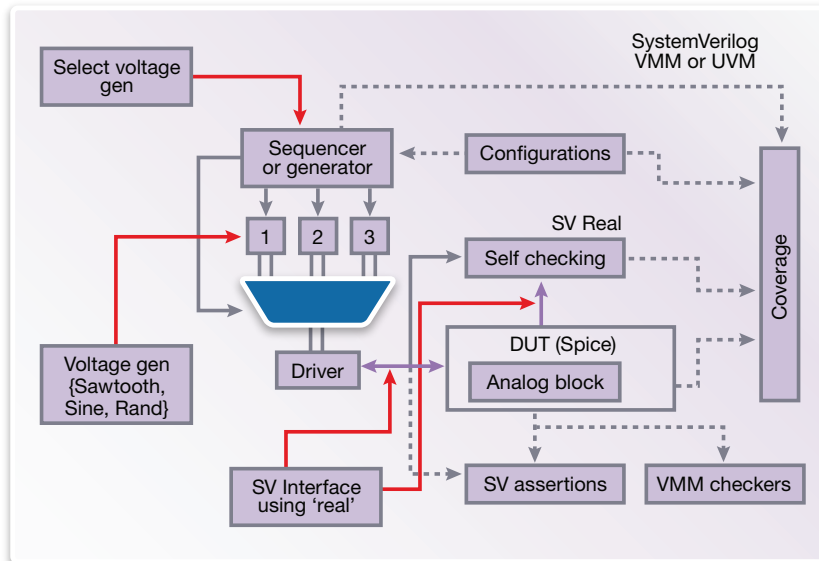


Figure 3: VCS AMS Testbench for Analog IP

AMS Assertions

VCS AMS testbench technology makes it possible to write assertions with digital or analog nodes. The latter usually trigger events that are necessary for creating immediate, concurrent assertions or sequences.

Immediate Assertions

Analog assertions can be modeled with immediate assertions that consist of a sampling event and a property to be verified. The sampling event is usually a digital clock and the property is a combination of expressions applied to SystemVerilog real scalars. As shown in Figure 4, properties can simply be modeled as synchronous immediate assertion which checks whether an analog node remains below 1.8V on each rising edge of clk.

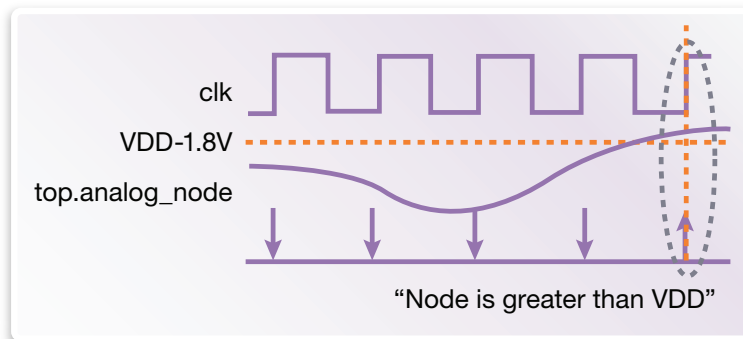


Figure 4: Analog Assertion

This can typically be used to make sure a DUT voltage reference always remains under a given value. For example, the following pseudo code shows how to verify the above property:

```
always @(posedge clk)
assert(top.analog_node <= 1.8)
else $error("Node is > VDD");
```

Note that you can write assertions which are asynchronous in nature by using analog events.

Concurrent Assertions

Concurrent assertions are used to check more complicated behavior. These are statements that assert that the specified properties must be true. Such properties are usually needed for verifying well-defined protocols or behaviors. For example, a PLL should be locked to the main frequency after a given timeframe that can be expressed in terms of clock cycles. Another example would be to verify relations between analog nodes. For instance, when *i* (analog node of interest) value is bigger than 90% of $V_{DD} = 1.80V$, *z* must be above 90% of V_{DD} on the next clock or the following clock.

To assert such a property the assertion can be written as follows:

```
assert property @(posedge clk)
```

```
(i>1.62) |-> ##[1:2] (z>1.62);
```

Sequences

Analog assertions can be explicitly specified in a sequence by using the non-overlapped operator `|->`, where subsequent sequences are evaluated one after the other after each analog event. The following example shows how to write a non-overlapped implication, the first element of the *s* sequence expression is evaluated on the next occurrence of `top.analog_clk`:

```
wire clk;
```

```
assign clk = top.analog_clk;
```

```
sequence s;
```

```
@(posedge clk)
```

```
(i>V_HIGH) |-> (z>V_HIGH);
```

```
endsequence
```

```
property p;
```

```
(a<V_LOW) |-> s;
```

```
endproperty
```

```
assert property(p);
```

Analog assertions can be explicitly specified in a sequence by using an overlapped operator `|=>`, where subsequent sequences are evaluated one after the other during the same analog event. The following example shows how to write an overlapped implication:

```
property p;
```

```
@(posedge clk)
```

```
(i>V_HIGH) |=> (z>V_HIGH);
```

```
endproperty
```

```
assert property(p);
```

Here, condition *i* is first evaluated. If it is true, then the *z* condition is evaluated at the same time.

Analog Events

The previous section explained how to trigger assertions with a digital clock. Though the triggering event was an analog event, it was converted to a Verilog wire, making it easier to trigger sequences. It is also possible to directly use the analog event as the triggering event:

```
always @(analog_clk)
assert(top.analog_node <= 1.8)

else $error("Node is > VDD");
```

AMS Checkers

As VCS AMS testbench technology provides the foundation for implementing analog assertions, it is natural to define some common analog checkers. This way, popular checkers can be modeled as Verilog modules that can be easily instantiated and bound to any analog node. The technology also provides these checkers modeled as transactors extending from `vmm_xactor` and `uvm_component`. The main advantage of these components is that they can be controlled from the overall VMM/UVM environment. It becomes possible to decide when to start or stop them and get them implicitly controlled with other components.

Threshold Checker

There are many cases where one would like to ensure an analog signal remains within a given range. For instance, some outputs should never go above a given voltage, else it might destroy the subsequent stages while overshooting. Another example could be when an input of a given stage becomes negative or superior to a given threshold; this situation could occur when two stages have different power domains and there are no adequate level shifters.

VCS provides a checker that verifies that an analog node remains within a given high and low threshold. This check can be performed synchronously or asynchronously, as illustrated in Figure 5. The first checkpoint is valid as the voltage is within the expected ranges; however, the second checkpoint will fire off an error as the voltage becomes higher than expected.

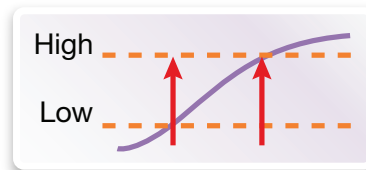


Figure 5: Threshold Waveform

Window Checker

There are situations where one would like to ensure an analog signal remains stable with a given tolerance. For instance, voltage references or band gaps should continue sinking a stable voltage reference independently of process, supply voltage and temperature. VCS provides a checker that verifies an analog node remains stable below or above a given threshold, as illustrated in Figure 6. The first checkpoint is invalid as the voltage is not within the expected tolerances, therefore, the checker will fire off an error.

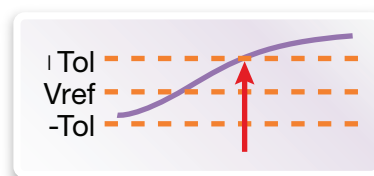


Figure 6: Window Checker

Slew Rate Checker

There are situations where one would like to ensure the slew rate of an analog signal remains below or above a given value. For instance, comparators have a minimal output slew rate that must be respected independently of process, supply voltage, and temperature. VCS provides a checker that verifies an analog node slew rate (\pm tolerance) remains below or above a given value, as illustrated in Figure 7. The first checkpoint is valid as the voltage slew rate is greater than expected dV/dt ; however, the checker will fire off an error on the second checkpoint.

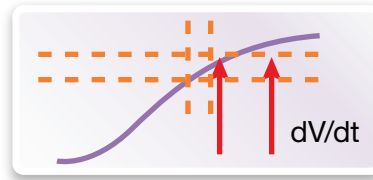


Figure 7: Slew Rate Checker

Frequency Checker

There are situations where one would like to ensure the frequency of an analog signal remains within a given tolerance. For instance, PLLs are supposed to output stable frequency once they are locked and this independently of process, supply voltage, and temperature. VCS provides a checker that verifies an analog node frequency remains stable (\pm tolerance), as illustrated in Figure 8. The first checkpoint is valid as the voltage frequency is as expected; however, the checker will fire off an error on the fourth checkpoint as the period decreased.

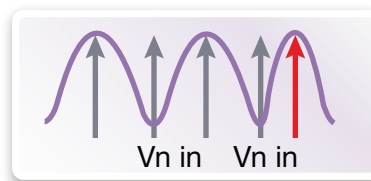


Figure 8: Frequency Checker

Voltage References

The VCS AMS testbench technology contains a few source generators that can be bound to internal or external analog nodes. These generators are responsible for driving a particular traffic to the analog IP. For example, the sine source generator can drive a well-defined sine with determined minimum/maximum voltages at a given frequency. Other source generators, such as sawtooth and square are also available. Injection of random voltage is also possible. As shown in the following diagram, it is possible to pick a generator followed by another one, and so on.

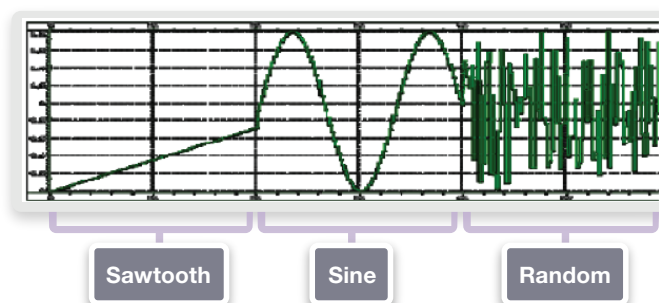


Figure 9: Multi-generation Modulation

Additionally, source generators can be combined to inject more complicated traffic. A typical situation is to add white noise on top of a carrier, which is generated with a sine generator. This can be useful to model external perturbation or to determine the common-mode rejection ratio (CMRR) of a differential amplifier (or other device), which is the tendency of the device to reject input signals common to both input leads.

As shown in Figure 10, the signal of interest can be represented by a small voltage fluctuation superimposed on a voltage carrier.

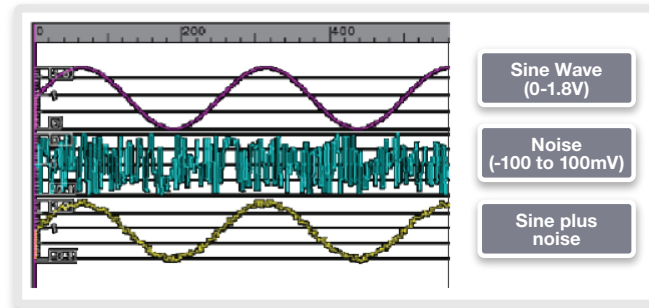


Figure 10: Carrier with Noise

As these generators do output voltages at a regular pace, it is important to ensure the signal shape is accurate enough without slowing down the simulation too much. An acceptable tradeoff is to output 10 samples per period (e.g., sample rate = 10X frequency).

Standard Voltage Generator

It is typical in SPICE to drive analog nodes with a sawtoothed or triangle shape, which can be done with the SAWGEN directive. VCS proposes a sawtooth source generator that provides a real value which can be used to drive analog voltage node with a sawtooth shape. You can specify min. and max. voltages V_{min} , V_{max} and the frequency f , as shown in Figure 11.

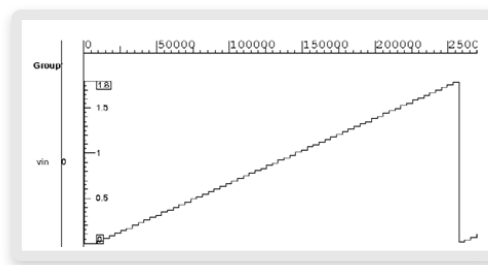


Figure 11: Sawtooth Waveform Generator

Another very important generator is the sine waveform generator. VCS-AMS provides a real value waveform generator that can be used to drive analog voltage node with a sine shape. You can specify min. and max. voltages V_{min} , V_{max} and the frequency f . These values can be changed during the simulation time. This is shown in Figure 12

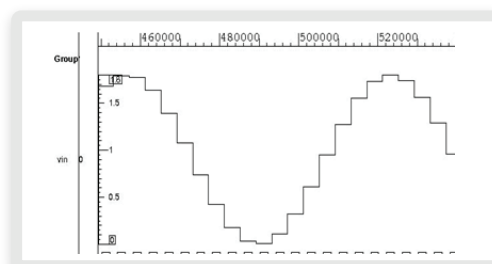


Figure 12: Sine Waveform Generator

Similarly, VCS provides a real value waveform generator that can be used to drive analog voltage node with a square shape (which can be done with the PWL directive in SPICE). You can specify min. and max. voltages V_{min} , V_{max} , the frequency f and the duty cycle. These values can be changed during the simulation time. (See Figure 13.) The square voltage source generator provides a real value that can be used to drive analog voltage node with a square shape.

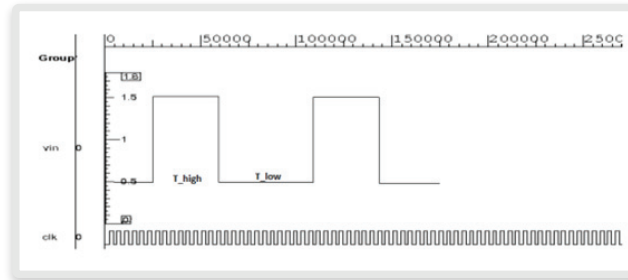


Figure 13: Square Waveform Generator

In addition, VCS provides a random voltage generator that provides a real value that can be used to drive analog voltage node with a random shape. This can be useful when combined with other standard source generators. For instance, it can be used to inject errors, distortion or perturbation to a given known good signal. It can also be used for adding noise on top of a carrier or directly in a voltage reference. You can specify min. and max. voltages V_{min} , V_{max} . These values can be changed during the simulation time. (See Figure 14.)

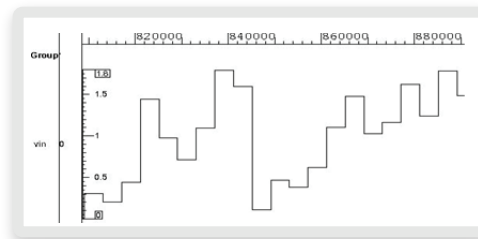
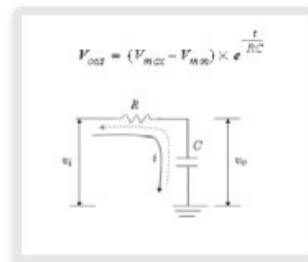


Figure 14: Random Source Generator

Custom Voltage Generator

In addition to predefined voltage shapes, it is possible to write custom voltage source generators. For example, to model a RC low-pass filter governed by the following equation:



This generator can then be controlled and initialized to output voltage as illustrated in Figure 15, where $V_{min} = -1V$, $V_{max} = 1V$, $f = 1MHz$, $RC = 200ns$:

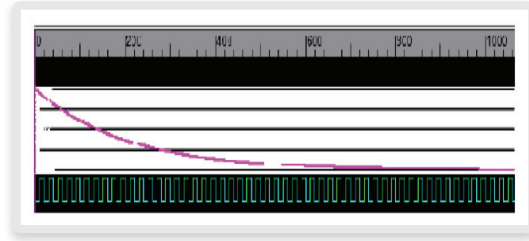


Figure 15: RC Waveform Generator

For more details, refer to the VCS user guide.

Voltage Generator Control

As described in the previous sections, all the source generators are easily constructed and their analog value is deposited to analog node by explicitly calling their `get_voltage()` method at regular times. This use model is easy but can be further simplified with the generic VMM source generator, which is based on `vmm_xactor`. This technique provides better integration, direct access to predefined SystemVerilog interface and better controllability (hence reuse at SoC for example). Additionally, this component comes with predefined notification that indicates when its embedded source generator reaches its half period. This is useful for changing its parameters in a well-defined way, i.e., when the source generators reach a given state. Furthermore, the source generator can be started and stopped on purpose by using the VMM `start_xactor` and `stop_xactor` methods.

Case Study

To prove that a generic CustomSim and VCS mixed-signal solution built on top of VMM or UVM would be viable, a simple amplifier DUT and its verification environment were developed. The objective of the case study was to replace Calvatec's existing SystemVerilog testbench and the AMS base classes by using a more mainstream methodology, such as Synopsys' VCS AMS testbench technology built on the Accellera UVM code.

The criterion for the case study was to leverage the predefined standard voltage generators, which in turn would leverage the built-in math functions. UVM was used as the underlying SV base class library and the simulations were executed in the following two modes.

- ▶ Mode 1: Pure digital simulation with VCS only.
- ▶ Mode 2: Mixed-signal simulation with VCS and CustomSim

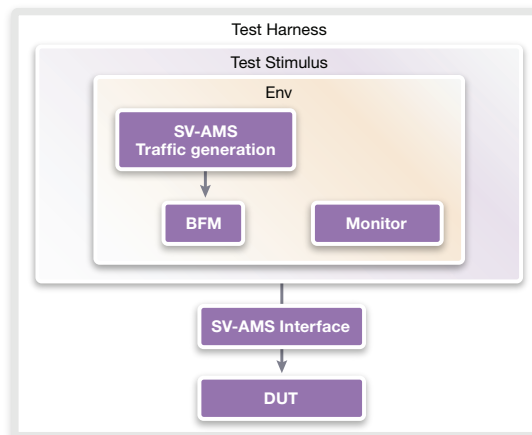


Figure 16: Architectural Overview

As shown in Figure 16, the SystemVerilog interface was used to carry the AMS signal information. The default AMS interface was placed in the UVM resource database then used by the verification environment to drive the signal into DUT and use the built-in checkers to ensure the signal was correct at the monitor. Adding more classes such as agents, tests, configurations, sequencers, etc., were superfluous to the objective of understanding how the SVAMS generation would verify DUT.

As shown in Figure 17, the cp_vmax analog voltage nodes can be broken down into ranges {min, typ, max} and crossed with another node cp_vi. This is an efficient way of verifying that all possible combinations are covered. In this case, we can see that cp_vi didn't fully address all expected values.

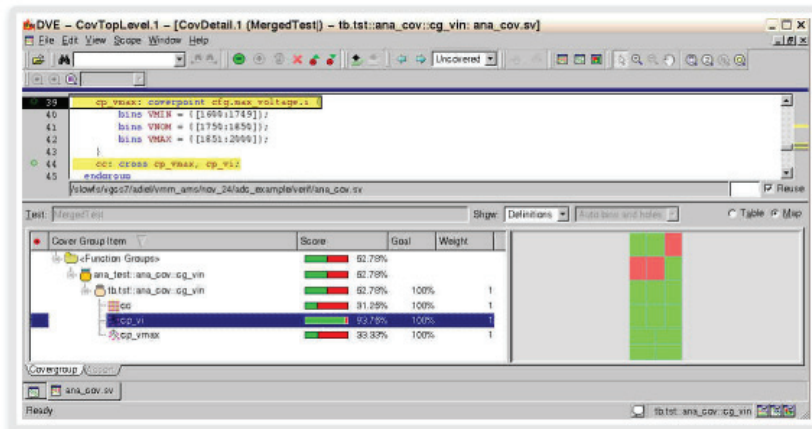


Figure 17: Analog Functional Coverage

Conclusion

By successfully proving that the setup and code runs as expected, the case study example could be extended to have multiple agents with multiple BFMs and monitors in the verification environment. Therefore, Calvatec will now have access to the complete benefits of UVM and the VCS AMS testbench technology without having to maintain its own internal base classes.

This also means they can leverage the built-in checkers and techniques used to verify mixed-signal DUTs.

The key advantage of having both the modes, pure digital and mixed-signal simulations, means that you can control the performance and accuracy of the simulations. For certain parts of verification it is perfectly legitimate to run with very low accuracy but high performance, enabling much better regression throughput. Then, by enabling more accuracy for other types of tests without a change to the infrastructure, establish confidence in other areas of DUT.

All of this coupled with the built-in coverage models delivered with VCS AMS testbench technology make it a robust solution offering increasing confidence in the design while mitigating risk.