VMM
Analog Mixed Signal
LCA Features User Guide

G-2012.09
September 2012

Comments?
E-mail your comments about this manual to:
vcs_support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2012 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AEON, AMPS, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignWare, EMBED-IT!, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, METeor, ModelTools, NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, Simply Better Results, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, Syndicated, Synplicity, the Synplicity logo, Synplify, Synplify Pro, Synthesis Constraints Optimization Environment, TetraMAX, UMRBus, VCS, Vera, and YIELDirector are registered trademarks of Synopsys, Inc.

Trademarks (™)

AFGen, Apollo, ARC, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, CustomExplorer, CustomSim, DC Expert, DC Professional, DC Ultra, Design Analyzer, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Eclypse, Encore, EPIC, Galaxy, HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIM^{plus}, i-Virtual Stepper, IICE, in-Sync, iN-Tandem, Intelli, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, Mars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Polaris, Power Compiler, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, Star-RCXT, Star-SimXT, StarRC, System Compiler, System Designer, Taurus, TotalRecall, TSUPREM-4, VCSi, VHDL Compiler, VMC, and Worksheet Buffer are trademarks of Synopsys, Inc.

Service Marks (^{sm})

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
Saber is a registered trademark of SabreMark Limited Partnership and is used under license.
All other product or company names may be trademarks of their respective owners.

Contents

4 AMS Checkers

5 AMS Source Generators

1

Introduction

Overview

The VMM Analog Mixed Signal (VMM-AMS) is a VMM application package used to verify analog IPs either at block level or at SoC level in an VMM environment.

The traditional verification approach used in analog world still lacks some key aspects that have been efficiently deployed to digital verification for years. SPICE-based analog verification environments are usually hard to reuse at SoC level, difficult to control and do barely bring the required simulation performances.

By leveraging from a well-proven VMM methodology, the main scope of VMM-AMS is to provide analog designers and verification engineers with a methodology that allows you to,

- Introduce analog verification planning
- Introduce constraint-random verification for driving analog nodes
- Model analog pin-wiggling 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 the above features, this guide also describes a scalable and reusable methodology for verifying analog IPs.

Reuse is made possible by correct modeling of verification models that can be stitched in SoC. These models can be implemented with HDL or Verilog-AMS, which depends upon the required accuracy.

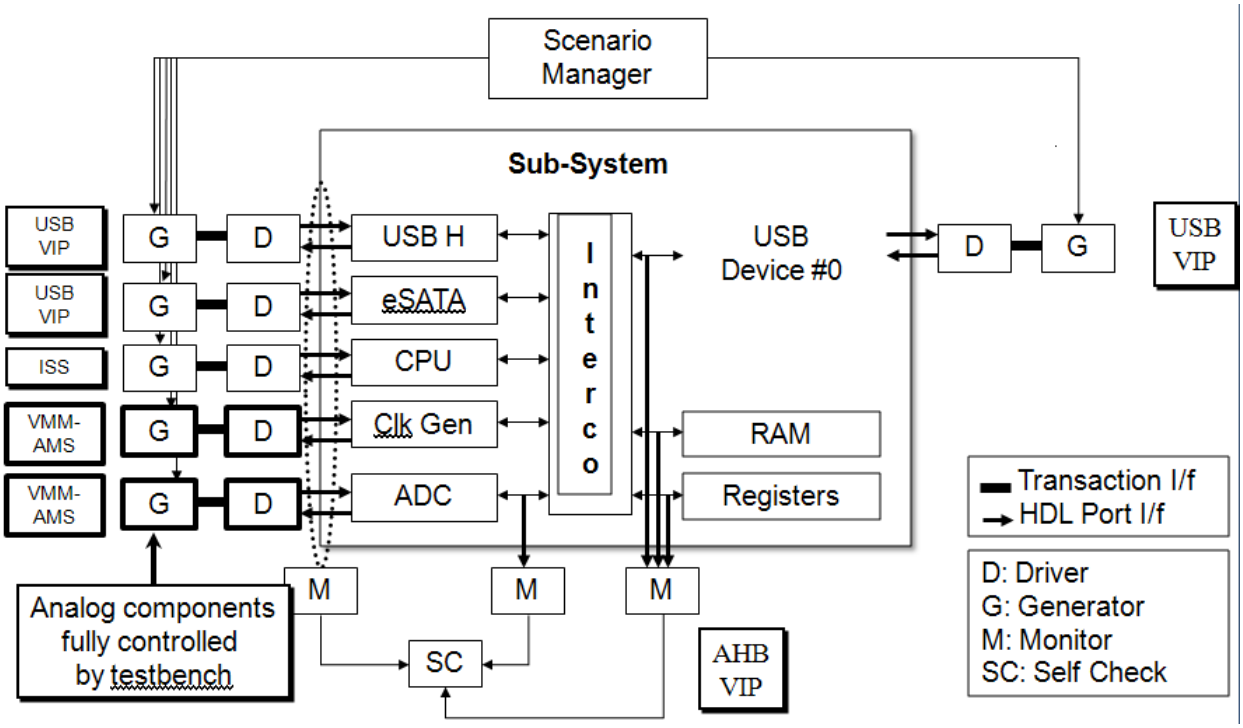
Architecture

New generation System on Chip (SoC) invariably increases the analog IPs included. For example, it is becoming usual to see multiple standard interfaces such as, USB, Ethernet, Sata, DDR, etc on a single chip. Additionally, there is a need for more analog IPs to handle multiple power domains, clock generation (PLL) and conversion (ADC, DAC).

As the need to include more embedded analog IPs increases, it is becoming challenging to architect verification environment that can accomadate digital and analog verification.

VMM-AMS provides a solution that helps to fill this gap. As shown in Figure 1-1, VMM-AMS allows to complement a traditional digital verification environment with a few components that can drive some analog IPs such as ADCs or Clock generation.

Figure 1-1 Using VMM-AMS Components for SoC Verification



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 this subsytem 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 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, 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.

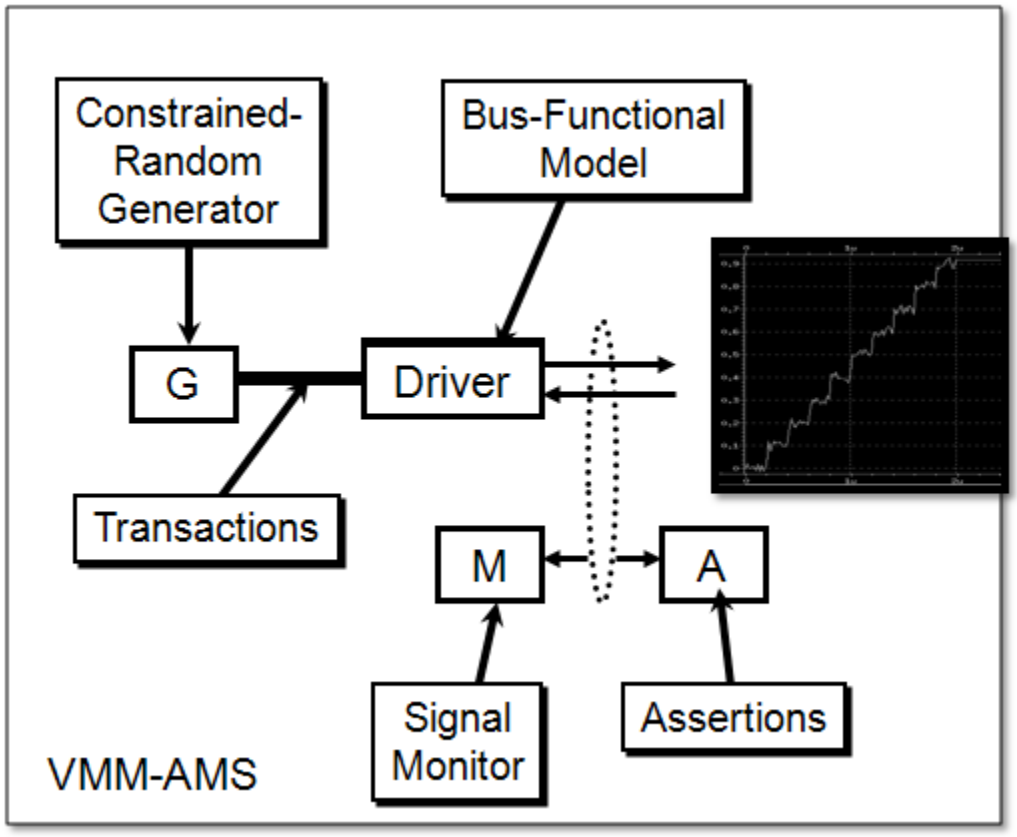
Note: SystemVerilog comes with wreal constructs that is usually used for fine-grain modelization of the analog behavior.

As shown in Figure 1-2, VMM-AMS comes with the built-in base classes that allows to drive analog inputs with given shapes such as, sine, sawtooth, square and white noise. These source generators can be combined together to create specific shapes. For example, to add a sine waveform with a given maximum/minimum voltage and frequency with well distributed noise.

You can also use these base classes to model your own traffic shapes. An interesting application is to directly inject voltage waveform 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 specific traffic shapes which is extremely difficult or impossible to achieve in SPICE.

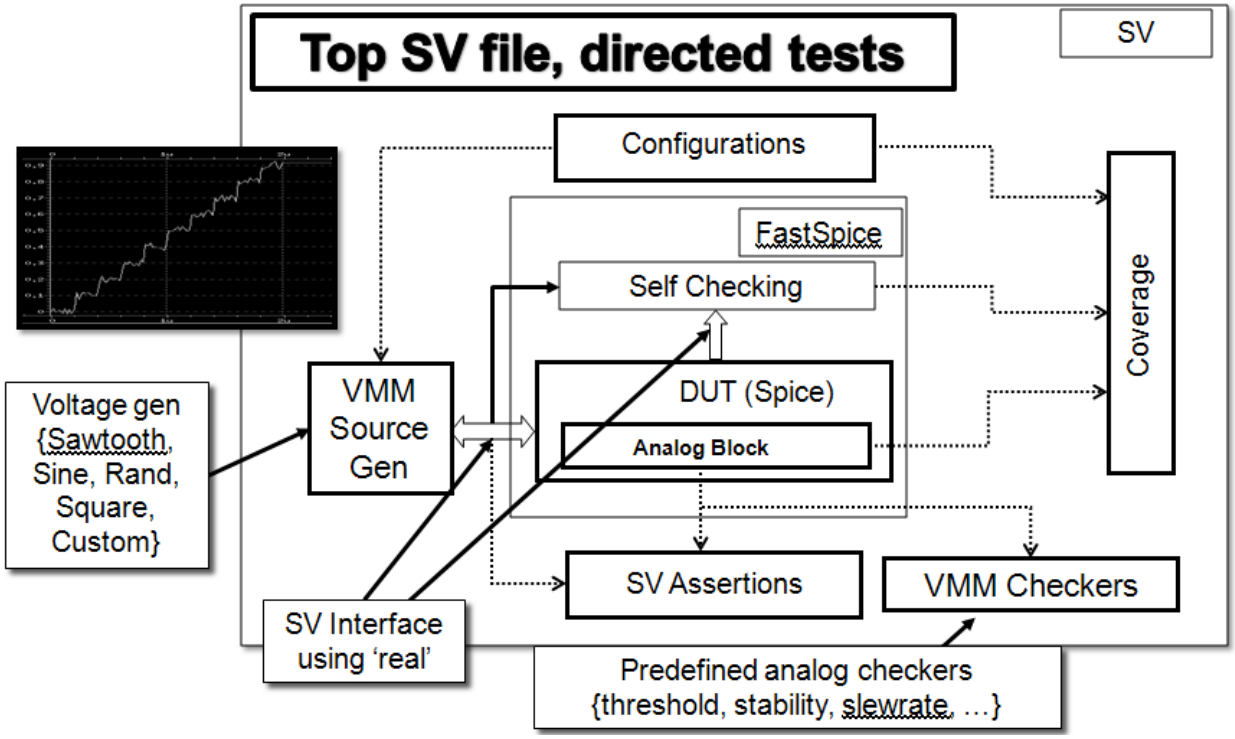
Figure 1-2 VMM-AMS Components



The key aspect of VMM-AMS is that the reuse of verification environment from block-level to system-level is becoming possible. As shown in Figure 1-3, VMM-AMS components can be used to drive the analog IP, sample its output and check whether its internal or external nodes do not violate some pre-defined rules.

As analog IP nodes can be converted to logic nodes, these logic nodes can be covered to ensure the toggle from 0->1 and 1->0 direction. Therefore, by using this mechanism it is possible to define analog toggle coverage. Toggle coverage is particularly useful for ensuring all analog nodes that are being accessed or exercised, therefore mimizing the potential interconnectivity issues.

Figure 1-3 VMM-AMS for Block-Level Verification



How to Use This Guide?

The main purpose of VMM-AMS is to build a verification environment in SystemVerilog language.

This guide is organized in multiple chapters that can read independently. These chapters cover the following methodology aspects:

- Chapter 2, "Mixed-AMS Co-Simulation" covers the VMS-AMS foundation for building efficient co-simulation between your verification environment and analog IP.
- Chapter 3, "AMS Assertions" explains how to write analog assertions to check protocol consistency of your analog DUT.
- Chapter 4, "AMS Checkers" explains how to use pre-defined checkers to check protocol consistency of your analog DUT.
- Chapter 5, "AMS Source Generators" explains how to drive your analog DUT with pre-defined voltage source generator.

2

Mixed-AMS Co-Simulation

This chapter provides guidelines for taking advantage of an efficient mixed-AMS co-simulation between your verification environment and analog DUT. It also covers the VMM-AMS foundation for building efficient co-simulation between your verification environment and analog IP.

Overview

The main purpose of the VMM-AMS methodology is to provide guidelines, foundation and base classes to build a verification environment in SystemVerilog language that allows efficient and reusable co-simulation with your analog IP.

The following topics are covered in this chapter:

- Bundling your analog IP nodes in a reusable SystemVerilog interface
- Handle toggle coverage of analog nodes (either voltages or current)
- Driving and sampling your analog DUT from SystemVerilog to a SPICE and/or Verilog-AMS
- Synchronous and asynchronous sampling of the analog nodes of your IP from SystemVerilog to a SPICE and/or Verilog-AMS
- Exchange information with other components, either analog or digital

Connecting Your Testbench to DUT

Bundling Analog Nodes in Interfaces

The SystemVerilog language provides an efficient and reusable way of grouping signals by using `interfaces`. You can simply declare your analog and digital signals in this object. In addition, you can create `clocking` blocks for defining signal directions and synchronization, and create `modports` for defining asynchronous associations. See IEEE-P1800 for further information.

Example 2-1 shows how to model the signals of an 8-bit ADC where `vin` is its analog input, `clk` is its sampling clock and `out` is its 8-bit output bus. The `clocking` block `dck` is used to gather `vin` and `out` signals as output and input respectively, these signals are synchronous upon `clk`. The modport `drive` provides a synchronous access to `sck`, which is usually used by the testbench

to drive the ADC analog input and sample its digital output. The modport `sample` provides a synchronous access to `sck`, which is usually used by the testbench to sample the ADC. The modport `async_sample` provides asynchronous access to the ADC signals, which is usually used to monitor the access to the ADC.

Example 2-1 Analog Signals in SV Interface

```
interface ana_comp_if(input bit clk);
   real vin;
   logic clk;
   logic[7:0] out;

   clocking dck @(posedge clk);
      output vin;
      input out;
   endclocking: sck

   clocking sck @(posedge clk);
      input vin;
      output out;
   endclocking: sck

   modport drive(clocking dck);
   modport sample(clocking sck);
   modport async_sample(input clk, input vin,
                        output out);

endinterface: ana_comp_if
```

Instantiation of Analog IP and Testbench Connection

This section explains how to connect your analog IP to your testbench. The analog IP can either be in Verilog-AMS or SPICE language. The testbench should be in SystemVerilog as it provides all the foundations for effectively verifying your IP. This approach will certainly help you to verify SoC with embedded analog IPs.

As explained in “Bundling Analog Nodes in Interfaces” , all analog signals should be modeled in a SystemVerilog interface. Then this interface can simply be shared between the SystemVerilog testbench and the analog IP.

The next set of examples explain how to model the interface, write a very simple random voltage generator and connect the testbench to the analog IP.

Example 2-2 shows how to model the signals of an 4-bit ADC where `vin` is its analog input, `s` is its sampling clock and `o1-o4` are the 4-bit output bus.

Example 2-2 4-Bit ADC Analog Signals in SV Interface

```
interface ana_comp_if(input bit clk);
  real vin;
  real s;
  logic o1, o2, o3, o4;
  ...
endinterface
```

Example 2-3 shows how to model a very simple test that randomize the ADC input signal. The analog interface `ana_if` is passed to the module `test` as argument, and the `vin` signal is assigned with a random value that is comprised in the [0:1.8V] range. VMM-AMS provides more scalable solutions for driving analog signals that are discussed in “Voltage Source Generators” on page 8.

Example 2-3 4-Bit ADC Analog Signals in SV Interface

```
module test(ana_comp_if ana_if);
  integer cnt=0;
  integer start=0;

  // Driver
  initial begin
     forever begin
```

```
        @(negedge ana_if.clk);
        ana_if.vin = ($urandom() & 1800)/1000.0;
        cnt++;
       if(cnt>50) $finish;
    end
   end
endmodule
```

Example 2-4 shows how to instantiate the module test, the ADC and connect them together using the analog interface `ana_if`.

Note: This file also contains both digital and analog clock generation of signals `clk` and `s` respectively.

Example 2-4 4-Bit ADC Analog Signals in SV Interface

```
module tb;
  logic clk=1'b0;
  ana_comp_if ana_if(clk);

  A2DConverter DUT(ana_if.vin, ana_if.s,
                   ana_if.o1, ana_if.o2,
                   ana_if.o3, ana_if.o4);
  test tst(ana_if);

  always #10 clk = ~clk;
  always @(clk) begin
    ana_if.s = (clk) ? 1.8 : 0.0;
  end
endmodule
```

Driving and Sampling Analog Nodes

The concept of VMM-AMS is based on the principle of Verilog/ SystemVerilog having access to internal analog nodes for read and write operations. With such an access, you can also apply similar VMM techniques used in the digital Verification realm to analog nodes.

VMM-AMS provides the following two different ways to access the analog nodes:

1. Accessing internal analog nodes as logic values. This aspect is covered in “Logic To Electrical Cross-References” .
2. Accessing internal analog nodes as real values. This aspect is covered in “Real To Electrical Cross-References” .

Logic To Electrical Cross-References

VMM-AMS provides system functions that allows to convert electrical from/to SystemVerilog logic. This capability is necessary for efficiently driving or sampling analog nodes, especially when there is no need of accuracy and that a conversion to 0 or 1 is enough. This conversion also enables toggle coverage of analog nodes. Low and high threshold can be customized to determine when an analog node should be considered as 0 or 1.

The logic mixed-language cross-reference method resembles conventional XMR (Cross Module Referencing) in SystemVerilog. In this method, the SystemVerilog code can simply treat an internal analog node as a logic value for read or write operations.

You must use the same XMR principles used to access a digital net to access an analog net. This means that the complete hierarchical path to the analog node must be given whenever an XMR read or write is operated.

VMM-AMS inserts a2d or d2a converters automatically depending on whether the Verilog is reading from or writing to the internal analog node. The inserted converters are subject to the same rules that govern the conventional interface nets including resistance map lookup.

These a2d and d2a converters appears in the "simv.msv/interface_element.rpt" file along with all the other interface elements. Similar to any other interface element, you can use the "a2d" and "d2a" mixed-signal commands to change the default settings for the a2d or d2a converters inserted for logic XMR.

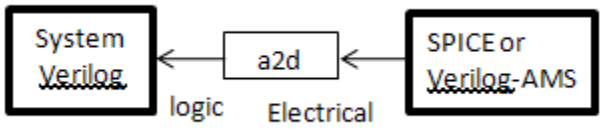
The following sections explain how to perform “Logic Read Access To Analog Nodes” and “Logic Write Access To Analog Nodes” .

Logic Read Access To Analog Nodes

This section explains how your testbench can access an analog node in write mode (to convert an analog node to a logic value).

As shown in Figure 2-1, automatic a2d is inserted between SystemVerilog and SPICE to allow the electrical to logic conversion.

Figure 2-1 Logic Read-Only Access of Analog Node



Example 2-5 shows an allocation of a logic XMR read on an analog node with the hierarchical path `top.i1.i2.x1.clk` into a SystemVerilog wire called `verilog_wire`.

The wire `verilog_wire` is asserted whenever `top.i1.i2.x1.clk` rises above a high threshold, it becomes deasserted when `top.i1.i2.x1.clk` falls below a low threshold.

Example 2-5 Verilog Wire XMR Assigned to an Analog Node

```
assign verilog_wire = top.i1.i2.x1.clk;
```

Example 2-6 shows an allocation of a logic XMR read on an analog node with the hierarchical path `top.i1.i2.x1.strb` into a SystemVerilog register.

Example 2-6 Verilog Register XMR Assigned to an Analog Node

```
initial begin
  verilog_reg = top.i1.i2.x1.strb;
  ...
end
```

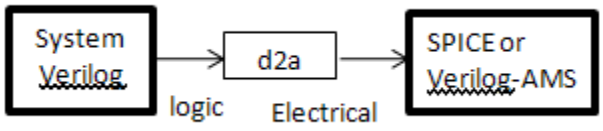
The register `verilog_reg` gets assigned a 1'b1 value whenever `top.i1.i2.x1.clk` rises above a high threshold, it gets assigned a 1'b0 value when `top.i1.i2.x1.clk` falls below a low threshold.

Logic Write Access To Analog Nodes

This section explains how your testbench can access an analog node in write mode, i.e. how to deposit a logic value to an analog node.

As shown in Figure 2-2, automatic d2a is inserted between SystemVerilog and SPICE to allow the logic to electrical conversion.

Figure 2-2 Logic Read-Only Access of Analog Node



Example 2-7 shows how a logic XMR write can be done on an analog node. The d2a converters inserted by VMM-AMS translates the logic values to voltage values and then apply them to the analog node.

Example 2-7 Insertion of d2a Between Verilog Register and Analog Node

```
reg rst_reg;
assign top.i1.i2.x1.rst = rst_reg;

initial begin
  …
  rst_reg = 1'b0;
  #5 rst_reg = 1'b1;
  …
end
```

In this example, the register `rst_reg` gets assigned a 1'b0 value which is converted to VSS on `top.i1.i2.x1.rst` analog node. 5ns after, `rst_reg` gets assigned a 1'b1 value which is converted to VDD on `top.i1.i2.x1.rst` analog node.

Real To Electrical Cross-References

VMM-AMS provides system functions that allows to convert electrical from/to SystemVerilog real. This capability is necessary for efficiently driving or sampling analog nodes with high accuracy, making it possible to deposit or sample a voltage or current on an analog node at a given point of time.

Note: This acts as a continuous assignment on a discrete system. for example, the value is deposited at a pre-defined pace such as a clock or sampling event.

The Real to Electrical cross-reference method resembles conventional XMR (Cross Module Referencing) in SystemVerilog. In this method, the SystemVerilog code can simply treat an internal analog node as a real value for read or write operations.

You must use the same XMR principles used to access a digital net to access an analog net. This means that the complete hierarchical path to the analog node must be given whenever an XMR read or write is operated.

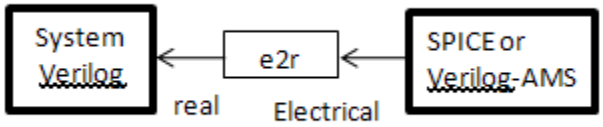
This method depends on the implementation of system functions and AMS-specific constructs to allow Verilog/SystemVerilog access and manipulate analog voltages and currents. The purpose is to mimic features available in the digital domain for AMS verification.

The following section explains how to perform “Real Read Access To Analog Nodes” and “Real Write Access To Analog Nodes” .

Real Read Access To Analog Nodes

As shown in Figure 2-3, automatic e2r is inserted between SystemVerilog and SPICE to allow the electrical to real conversion.

Figure 2-3 Logic Read-Only Access of Analog Node



VMM-AMS allows you to convert voltages and currents to SystemVerilog real. Here is the list of system functions that provide real-valued read access to analog nodes:

```
$snps_get_volt(analog_hier_node_name)
$snps_get_port_current(analog_hier_node_name)
```

Example 2-8 shows how to convert the voltage value of `top.i1.ctl` to a SystemVerilog real value stored in `r`, where `top` is the testbench name, `i1` is a SPICE module and `ctl` is the voltage node to read.

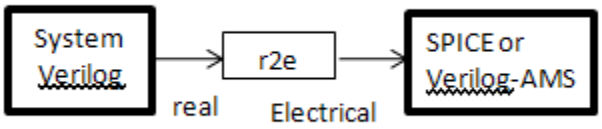
Example 2-8 Conversion of SPICE Node Voltage to SystemVerilog Real

```
real r;
always @(posedge clk)
   r <= $snps_get_volt(top.i1.ctl);
```

Real Write Access To Analog Nodes

As shown in Figure 2-4, automatic r2e is inserted between SystemVerilog and SPICE to allow the real to electrical conversion.

Figure 2-4 Logic Read-Only Access of Analog Node



VMM-AMS allows to force or release analog node voltages with zero delay. Here is the list of system functions that provide real-valued write access to analog node voltages:

```
$snps_force_volt(analog_hier_node_name,real_value)
$snps_release_volt(analog_hier_node_name)
```

Example 2-9 shows how to force the SystemVerilog real value stored in `r` to the `top.i1.ctl` analog node. It also shows how to release this value so that SPICE can possibly drive another value.

Example 2-9 Conversion of SPICE Node Voltage to SystemVerilog Real

```
real r;
always @(posedge clk)
   $snps_force_volt(top.i1.ctl, r);
   ...
   $snps_release_volt(top.i1.ctl);
```

Note: When a SystemVerilog real is bound to an analog node, it is possible to directly deposit the real value as a voltage to this node.

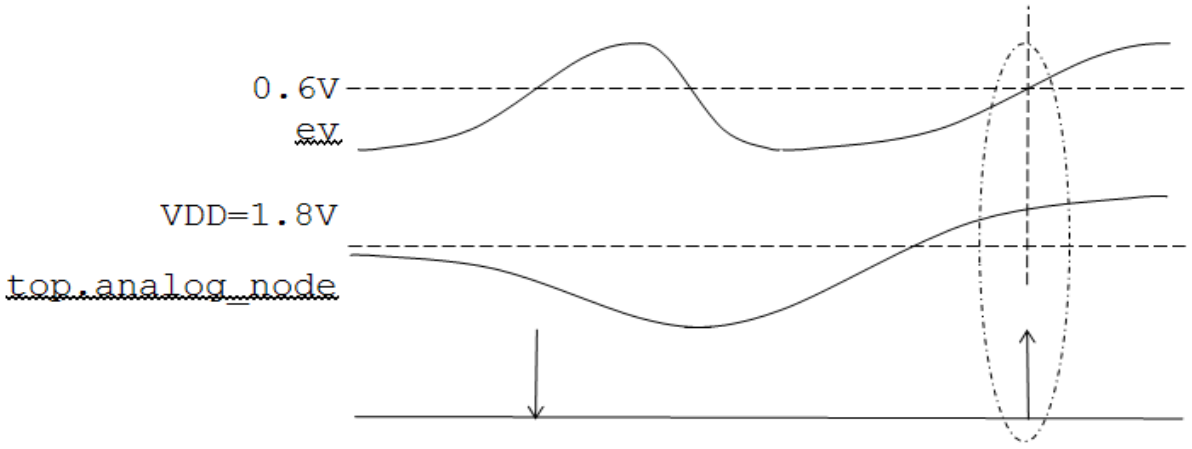
Synchronous and Asynchronous Sampling

As explained in previous sections, there is no continuos assignment between electrical and real/logic. This is simply because the electrical is handled by the Fast SPICE simulator and the logic/real is handled by the digital simulator. Both simulators have different schemes and the digital simulator is cycle-based. This combination allows the co-simulation to be effective.

There are many scenarios where electrical to logic or electrical to real conversion must take place. This conversion can either be synchronous or asynchronous. The synchronous operation is usually triggered with Verilog clocked `@always` blocks. The asynchronous operation can be triggered from the Fast SPICE simulator to the digital simulator. The latter enables to trigger digital or analog sampling directly from a SPICE analog node.

As shown in Figure 2-5, the analog node `top.analog_node` could be compared to 1.8V whenever the analog signal `ev` rises above 0.6V. Using this approach, it becomes possible to model assertions for analog nodes and to trigger data sampling directly from an analog node as well. VMM-AMS provides a set of checkers that rely on this technique.

Figure 2-5 Analog Asynchronous Sampling



To make the verification analog event-driven, you can use the following two blocking system tasks:

```
@snps_cross ( expr with analog_access_systask/function
              [ , dir [ , time_tol [ , expr_tol ] ] ] ) ;

@snps_above ( expr with analog_access_systask/function
             [ , time_tol [ , expr_tol ] ] ] ) ;
```

You can specify whether the hierarchical analog node should cross or be above a given threshold, in a specific direction with a given tolerance. Both system tasks are similar to the `@cross` and `@above` functions that come with Verilog-AMS but are targeted to be used with SPICE netlists.

Example 2-10 shows how to sample the node `top.i1.ctl`, where `i1` is a SPICE module and `ctl` is the voltage node to be used as a trigger. This code block is executed whenever the voltage of `top.i1.ctl` rises above 0.6V.

Example 2-10 Asynchronous Sampling of an Analog Node

```
always @(snps_cross($snps_get_volt(top.i1.ctl)-0.6,1))
  begin …
```

Example 2-11 shows how to sample the node `top.i1.port3`, where `i1` is a SPICE module and `port3` is the current node to be used as a trigger. This code block is executed whenever the current of `top.i1.port3` falls below 1μA.

Example 2-11 Asynchronous Sampling of an Analog Node

```
always @(snps_cross($snps_get_port_current(
                                  top.i1.port3-1.0E-6,-1))
  begin …
```

Example 2-12 shows how to sample the node `top.i1.ctl`, where `i1` is a SPICE module and `ctl` is the voltage node to be uses as a trigger. This code block is executed whenever the voltage `top.i1.ctl` rises above 1.2V.

Example 2-12 Asynchronous Sampling of an Analog Node

```
always @(snps_above($snps_get_volt(top.i1.ctl)-1.2,1))
  begin …
```

3

AMS Assertions

This chapter provides guidelines for implementing efficient and portable mixed signal assertions.

Overview

SystemVerilog supports the following two kinds of assertions:

- Immediate, using `assert` construct
- Concurrent, using `assert property` construct

As described in Chapter 2, "Mixed-AMS Co-Simulation", it is possible to write assertions with either digital or analog nodes. The latter usually trigger events that are necessary for creating immediate, concurrent assertions or sequences.

You can achieve the digital trigger event by the automatic insertion of Connect Modules.

Immediate Assertions

Immediate assertions are procedural statements similar to if-statements with procedural then/else code blocks.

Note: The immediate assertions are commonly used with ‘else’ statement only, also called “fail statement”. In this case, the attached code block is executed when the assertion condition is violated.

Example 3-1 shows how to model an assertion that checks whether `top.ev` remains below 1.8V on each rising edge of `clk`.

Example 3-1 Synchronous Immediate Assertion of Analog Node

```
always @(posedege clk)
  assert(top.ev <= 1.8)
  else $error("Node is greater than VDD");
```

To use an analog event for verifying such an assertion, see Example 3-2 to rewrite the previous assertion.

Example 3-2 Asynchronous Immediate Assertion of Analog Node

```
always @(snps_cross($snps_get_volt(top.ev)-0.6,1))
  assert(top.analog_node <= 1.8)
  else $error("Node is greater than VDD");
```

Note: `@snps_cross($snps_get_volt(voltage-threshold), direction)` is a Verilog function that triggers an event when voltage reaches threshold in a given direction. For details, see “Synchronous and Asynchronous Sampling” on page 13.

For performance reasons, the immediate assertions should be attached to an event. Although it is possible to model assert statements that continuously evaluate analog nodes coming from SPICE, the simulation performances will be too slow, and therefore, you should prefer the case analog events in this situation.

Concurrent Assertions

Concurrent assertions are used to check more complicated behaviors. These are statements that assert that the specified properties must be true.

For example, to check an expression, `top.analog_node <= 1.8` is never true at any point during simulation. Example 3-3 explains how to write this as an assertion.

This type of assertion can be used for verifying voltage references, power supply stability or signal overshooting.

Example 3-3 Checking Analog Node Value

```
assert property (top.analog_node <= 1.8);
```

Properties can be based on synchronous events and modeled in terms of sequences.

Example 3-4 shows how to write this as an assertion. Where, `i` is a simple sequence and `##[1:2] z` is a more complex sequence expression, meaning that when `i` value is bigger than 90% of VDD, `z` must be above 90% of VDD on the next clock or the following clock.

This type of assertion can be useful for verifying cause and effect of internal nodes in your analog IP. It is usually needed for verifying well-defined protocols or behaviors. For example, a PLL should be locked to the main frequency after a given time frame that can be expressed in terms of clock cycles.

Example 3-4 Synchronous Analog Assertion

```
assert property (@(posedge clk)
   (i>1.62) |-> ##[1:2] (z>1.62));
```

Implications

Assertions can be triggered either by a clock or an analog event by using the `@(snps_cross ...)` function.

The asynchronous event for a property can be specified in several ways:

- Non-overlapped implication with |-> operator
- Overlapped implication with |=> operator

Non-Overlapped Implication

In some cases, when the sequence of events are larger and it is appropriate to break down each cause-effect in terms of sequences.

Analog assertions can be explicitly specified in a sequence by using the non-overlapped operator `|->`, subsequent sequences are evaluated one after the other after each analog event.

As shown in Example 3-4, this type of assertion can be useful for verifying cause and effect of internal nodes in your analog IP.

Example 3-5 shows how to write an non-overlapped implication, the first element of the `s` sequence expression is evaluated on the next occurrence of `top.analog_clk`.

Note: The `top.analog_clk` node is converted to the `clk` Verilog wire so that it is easier to trigger these sequences.

Example 3-5 Non-Overlapped Implication in an Analog Assertion

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

sequence s;
 @(posedge clk)
    (i>VDD_H) |-> (z>VDD_H);
endsequence

property p;
    (a>VDD_H)|-> s;
endproperty

assert property (p);
```

Overlapped Implication

This will be the case when the analog event is explicitly specified in the property using the overlapped operator `|=>`.

Analog assertions can be explicitly specified in a sequence by using an overlapped operator `|=>`, subsequent sequences are evaluated one after the other during the same analog event.

Example 3-6 shows how to write a overlapped implication, `i` condition is first evaluated, if it is true then the `z` condition is evaluted at the same time.

Example 3-6 Overlapped Implication in an Analog Assertion

```
property p;
  @(posedge clk)
    (i>VDD_H) |=> (z>VDD_H);
endproperty

assert property (p);
```

Analog Events

The previous section explains 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.

Example 3-7 shows how to write an overlapped implication, `i` condition is first evaluated, if it is true then the `z` condition is evaluated at the same time. In this case, the assertion is triggered by the `analog_ev` node.

Example 3-7 Analog Event Explicitly Specified in Concurrent Assertion

```
assert property
  @(snps_cross($snps_get_volt(analog_ev)-THRESHOLD,1)
    (i>VDD_H) |=> (z>VDD_H);
```

Example 3-8 shows how to write the same assertion by decomposing into a property `p` and always block to trigger it when `analog_ev` node rises the above `THRESHOLD`.

Example 3-8 Concurrent Assertion With Property and Analog Event

```
property p
    (i>VDD_H) |=> (z>VDD_H);
endproperty

always @(snps_cross($snps_get_volt(analog_ev)-
THRESHOLD,1)
  assert property (p);
```

Deactivating Analog Assertions

There are few cases where an assertion should be deactivated, for example, when the analog IP is reset or in power down.

The `disable iff` clause allows an asynchronous analog event to shut down the assert statement evaluation.

In Example 3-9, if `analog_rst` reaches 90% of VDD during the evaluation of the sequence, then `p` is de-asserted. Else, if `power_down` is not asserted, then analog voltage reference node called `voltage_ref` should remain within 50% of VDD with a tolerance of 2.5%.

Example 3-9 Deactivating an Assertion

```
property p;
  @(posedge clk) disable iff (top.analog_rst>1.62)
                    !power_down
                ##1 (top.vref>=0.975 && top.vref<=1.025);
endproperty
assert property (p);
```

Reals in Assertions

You use reals in sequences and properties.

In Example 3-10, the variable `v` is assigned to `analog_in` on each clock. Five clocks later, `analog_out` is expected to be equal to `v`.

Example 3-10 Reals in Assertions

```
property delayed_value;
  real v;
  @posedge clk) (v=analog_in) ##5 (analog_out=== v);
endproperty
```

Mathematical Functions in Analog Assertions

You use mathematical functions in properties and sequences.

In Example 3-11, when `clk` rises, `vin` absolute value should be less than 1.8V and drop below 200mV after 100 clock cycles.

Example 3-11 Using Functions in Assertions

```
property stabilization;
  real v;
  always @(posedge clk)
    (abs(vin)<1.8) ##100 (abs(vin)<0.2)
endproperty
```

Binding

Assertions can be included directly in the source code of the modules in which they apply. Or they can be embedded in the procedural code of a module and then get bound to a SPICE netlist.

Assertions can be bound to the modules in the following language:

- SystemVerilog
- SPICE

- Verilog-AMS

The available SystemVerilog construct is as follows:

```
bind spice_subckt_name|spice_instance_name

   module_name [#(param_assignments)]

   instance_name [(list_of_port_connections)]
```

Where `spice_subckt_name` is the SPICE netlist where the module assertion should be bound, `module_name/ instance_name` are the SystemVerilog module and instance names.

4

AMS Checkers

VMM-AMS contains few checkers similar to OVL that can be bound to internal or external nodes. These checkers are responsible for ensuring a particular protocol which is always valid.

For instance, the threshold checker can fire-off an assertion failure whenever a given node reaches an invalid threshold.

These checkers are available as standalone module, written in SVA and as VMM transactor, written in SystemVerilog. Both the implementation are equivalent, the main difference is that checkers based on VMM transactor provide more flexibility in terms of reuse and controllability.

Table 4-1 provides a complete list of VMM-AMS checkers, which are written in SVA.

Table 4-1 SVA Checkers

ams_sync_threshold_checker	Checks that analog signal remains below or above a given threshold. Performs this check on each positive edge of clock
ams_sync_window_checker	Checks that analog signal remains within a given high and low threshold. Performs this check on each positive edge of clock
ams_sync_stability_checker	Checks that analog signal is stable (vref +/- tolerance) when enable is true
ams_sync_frame_checker	Checks that analog signal is stable (vref +/- tolerance) during a time window
ams_sync_slew_checker	Checks that analog signal rises/falls with a given slew rate(+/- tolerance)

Table 4-2 provides a complete list of VMM-AMS checkers, which are written in SystemVerilog and extends from `vmm_xactor` base class.

Table 4-2 VMM-AMS Checkers

vmm_ams_threshold_checker	Checks that analog signal remains below or above a given threshold. Performs this check synchronously or asynchronously
vmm_ams_frequency_checker	Checks that analog signal remains within a given frequency (+/- tolerance). Performs this check synchronously or asynchronously
vmm_ams_stability_checker	Checks that analog signal is stable (vref +/- tolerance) when enable is true. Performs this check synchronously or asynchronously
vmm_ams_slew_checker	Checks that analog signal rises/falls with a given slew rate(+/- tolerance). Performs this check synchronously or asynchronously

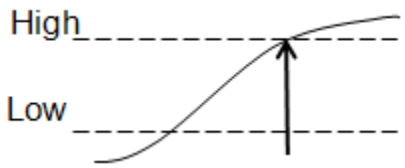
Threshold Checker

This module checks whether an analog node crosses a voltage threshold in a specified direction. For efficiency reasons, it uses a periodic clock to sample the signal.

This checker can be useful for verifying some analog components such as voltage references which remain below a given voltage for different processes, supply voltage, temperature, etc.

As shown in Figure 4-1, the threshold checker can be initialized with a constant threshold and constant direction.

Figure 4-1 Threshold Checker Diagram



SVA Threshold Checker

You can specify an absolute voltage threshold with the `.threshold` parameter and the direction with `.direction` parameter. As this module is synchronous, you need to specify a clock with `clk` input and an analog node `vin` to be checked.

```
ams_sync_threshold_checker #(.threshold(th),
                             .direction(dir))
   sync_rising_thr_chk(clk, vin);
```

Example 4-1 shows how to instantiate one `ams_sync_threshold_checker` modules to verify when the signal rises above 1.7V.

Example 4-1 Checking Analog Node Threshold Using SVA Checker

```
`include "vmm_ams.sv"
module top;
  logic clk=1'b0;
  real vin;

  // Checks the signal vin remains below 1.7V
  // Otherwise fires off an assertion failure
  ams_sync_threshold_checker #(.threshold(1.7),
                               .direction(+1))
     sync_rising_thr_chk(clk, vin);

  always #10 clk = ~clk;

endmodule
```

The output of Example 4-1 is,

```
"/vault/VCS_1006/etc/rvm/vmm_ams.sv", 356:
top.sync_rising_thr_chk.ams_sync_threshold_checker:
started at 990s failed at 990s
        Offending '(((direction == (+1)) && (vin <=
threshold)) || ((direction == (-1)) && (vin >=
threshold)))'
top.sync_rising_thr_chk.ams_sync_threshold_checker
Vin=1.76 Thr=1.70
```

VMM Threshold Checker

Similarly, VMM-AMS provides a threshold checker that is modeled using `vmm_xactor`. The main difference is that it can be explicitly started/stopped, its parameters can be changed during simulation. It supports either synchronous or asynchronous sampling.

Example 4-2 shows how to instantiate two `vmm_ams_threshold_checker` modules to verify when the signal rises above 1.65V or falls below 0.15V. Both the checkers constantly check the value of `ana_if.vin` which can be attached to an analog node or a real. The checking is performed asynchronously whenever `ana_if.vin` is changing. A message based on `vmm_log` is emitted whenever the checker detects a violation. By default, `vmm_error` is returned, it can be changed in the constructor using `vmm_log::WARNING_SEV` to dump a warning.

Note:In Example 4-2, both the checkers are explicitly started and stopped after 1000ns. Then, the positive checker gets its threshold changed to 1.7V.

Example 4-2 Checking Analog Node Threshold Using VMM Checker

```
module top;
  logic clk=1'b0;
  ams_src_if ana_if(clk);
  vmm_ams_threshold_checker#(1.65,+1,
            vmm_ams_generic_checker::ASYNCHRONOUS_REAL)
          pos_chk = new("PosThr", "", 0,

                        ana_if, vmm_log::WARNING_SEV);
  vmm_ams_threshold_checker#(0.15,-1,
             vmm_ams_generic_checker::ASYNCHRONOUS_REAL)
          neg_chk = new("NegThr", "", 0,
                        ana_if, vmm_log::WARNING_SEV);
  always #10 clk = ~clk;

  initial begin
     pos_chk.start_xactor();
     neg_chk.start_xactor();

     #1_000 pos_chk.stop_xactor();

     #1_000
     pos_chk.set_params(1.7,+1);
     pos_chk.start_xactor();
```

```
    end
endmodule
```

Window Checker

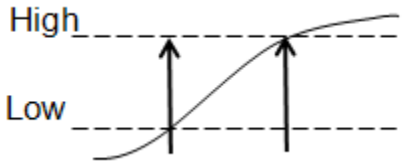
The windows checker module checks for the analog node to remain within lower and upper limits.

For efficiency reasons, it uses a periodic clock to sample the signal.

This checker is useful for verifying some analog components to remain within a given voltage range for different processes, supply voltage, temperature, etc. It can also check for harmful voltage overshoots, etc.

As shown in Figure 4-2, the threshold checker can be initialized with a constant threshold and a constant direction.

Figure 4-2 Window Checker Diagram



SVA Window Checker

You can specify an absolute voltage threshold low and high by using the `.threshold_lo` and `.threshold_hi` parameters. You can specify the analog node to be in or out of the window by using the `.mode` parameter (+1: in, -1: out).

As this module is synchronous, you need to specify a clock with `clk` input and an analog node `vin` to be checked.

```
ams_sync_window_checker #(.threshold_lo(th_lo),
                          .threshold_hi(th_hi),
                          .mode(mode))
sync_in_window_chk (clk, vin);
```

Example 4-3 shows how to instantiate `ams_sync_window_checker` module to verify for a signal that should remain between 0.1V and 1.7V.

Example 4-3 Checking Analog Node Window

```
module top;
  ...
  ams_sync_window_checker #(.threshold_lo(0.1),
                            .threshold_hi(1.7),
                            .mode(+1))
     sync_in_window_chk (clk, vin);
  ...
endmodule
```

The output of Example 4-3 is,

```
"/vault/VCS_1006/etc/rvm/vmm_ams.sv", 386:
top.sync_in_window_chk.ams_sync_window_checker: started at
990s failed at 990s
      Offending '(((mode == (+1)) && ((vin > threshold_lo)
&& (vin < threshold_hi))) || ((mode == (-1)) && ((vin <
threshold_lo) || (vin > threshold_hi))))'
top.sync_in_window_chk.ams_sync_window_checker Vin=1.76
Mode=1 Thr=[0.10-1.700000]
```

VMM Window Checker

Similarly, VMM-AMS provides a window checker that is modeled using `vmm_xactor`. The main difference is that it can be explicitly started/stopped and its parameters can be changed during simulation. It supports either synchronous or asynchronous sampling.

It is based on the **`vmm_ams_threshold_checker`** that provides a general purpose implementation. See “VMM Threshold Checker” for more details.

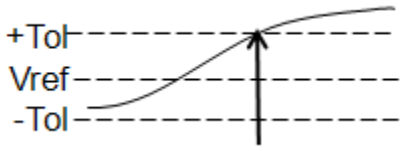
Stability Checker

The stable checker module checks an analog node to remain stable within a given tolerance when it is enabled.

This is useful for verifying some of the analog components such as power management stability when activated within a range for different processes, supply voltage, temperature, etc.

As shown in Figure 4-3, the stability checker can be initialized with a constant reference and tolerance.

Figure 4-3 Window Checker Diagram



SVA Stability Checker

You can specify an absolute voltage value by using `.vref` and tolerance by `.tolerance` parameters.

```
ams_sync_stable_checker#(.vref(vref),
                         .tolerance(tol))
   sync_rising_thr_chk (clk, vin, ena);
```

As this module is synchronous, you need to specify a clock with `clk` input and an analog node `vin` to be checked, this module is activated when the `ena` signal is asserted.

Example 4-4 shows how to instantiate `ams_sync_stable_checker` module to verify for a signal to remain stable to 1.0V with a tolerance of +/- 7.5%.

Example 4-4 Checking Analog Node Window

```
module top;
  ...
  ams_sync_stable_checker#(.vref(1.0),
                           .tolerance(0.075))
     sync_rising_thr_chk (clk, vin, ena);
  ...
endmodule
```

The output of Example 4-4 is,

```
"/vault/VCS_1006/etc/rvm/vmm_ams.sv", 386:
```

```
top.sync_in_window_chk.ams_sync_window_checker: started at
990s failed at 990s
      Offending '(((mode == (+1)) && ((vin > threshold_lo)
&& (vin < threshold_hi))) || ((mode == (-1)) && ((vin <
threshold_lo) || (vin > threshold_hi))))'
top.sync_in_window_chk.ams_sync_window_checker Vin=1.76
Mode=1 Thr=[0.10-1.700000]
```

SVA Frame Stability Checker

The frame checker module checks for an analog node to remain stable within a given time frame when it is enabled.

For efficiency reasons, it uses a periodic clock to sample the signal and a control signal that determines when the check must be performed.

This checker is useful for verifying some of the analog components such as power management stability when activated within a range for different processes, supply voltage, temperature, etc.

```
ams_sync_frame_checker#(.vref(vref),
                        .tolerance(tol),
                        .window_low(wlo),
                        .window_high(whi))
   sync_rising_thr_chk (clk, vin, ena);
```

You can specify an absolute voltage value by `.vref` and tolerance by `.tolerance` parameters. The analog node can be checked within a given time window as provided with `.window_low` and `.window_high` parameters.

As this module is synchronous, you need to specify a clock with `clk` input and an analog node `vin` to be checked, when the `ena` signal is asserted.

Example 4-5 shows how to instantiate `ams_sync_frame_checker` module to verify for a signal to remain stable at 1V, with a tolerance of +/-7.5%, between time 750ns and 1000ns.

Example 4-5 Checking Analog Node Window

```
module top;
  ...
  ams_sync_frame_checker#(.vref(1.0),
                          .tolerance(0.075),
                          .window_low(750),
                          .window_high(1000))
     sync_rising_thr_chk (clk, vin, ena);
  ...
endmodule
```

This output of Example 4-5 is,

```
"/vault/VCS_1006/etc/rvm/vmm_ams.sv", 439:
top.sync_rising_thr_chk.ams_sync_frame_checker: started at
990s failed at 990s
        Offending '((vin >= ((1 - tolerance) * vref)) &&
(vin <= ((1 + tolerance) * vref)))'
top.sync_rising_thr_chk.ams_sync_frame_checker Vin=1.10
Ref=1.00
```

VMM Stability Checker

Similarly, VMM-AMS provides a stability checker that is modeled using `vmm_xactor`. The main difference is that it can be explicitly started/stopped and its parameters can be changed during simulation. It supports either synchronous or asynchronous sampling.

Example 4-6 shows how to instantiate two `vmm_ams_stability_checker` modules to verify that `ana_if.vin` signal remains equal to 1.5V +/- 1%. It constantly

checks the value of `ana_if.vin` which can be attached to an analog node or a real. The checking is performed synchronously, i.e. on each positive edge of `clk`. A message based on `vmm_log` is emitted whenever the checker detects a violation. By default, a `vmm_error` is returned.

Note: In Example 4-6, the checker is explicitly started and stopped after 1000ns. Then, its voltage reference is changed to 1.45V and its tolerance changed to +/- 5%.

Example 4-6 Checking Analog Node Threshold Using VMM Checker

```
module top;
  logic clk=1'b0;
  ams_src_if ana_if(clk);
  vmm_ams_stability_checker#(1.5,0.01)
                       chk = new("Chk", "", 0, ana_if);
  always #10 clk = ~clk;

  initial begin
     chk.start_xactor();

     #1_000 chk.stop_xactor();
     chk.set_params(1.45,0.05);
     pos_chk.start_xactor();

  end
endmodule
```

This checker provides a general purpose implementation that allows to check both constant stability and windowed stability and therefore only one base class is necessary.

Slew Rate Checker

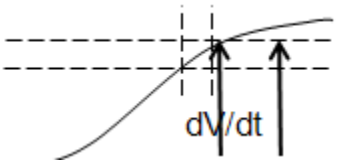
The slew rate checker verifies that the input signal is monotonically rising or falling within a specified dv/dt slew rate value.

For efficiency reasons, the SVA implementation uses a periodic clock to sample the signal and a control signal that determines when the check must be performed. The VMM implementation if fully controllable and also supports synchronous/asynchronous data sampling.

This checker can be useful for verifying some analog components, such as comparators are able to drive an output with a minimum slew rate for different capacitance, processes, supply voltage, temperature, etc.

As shown in Figure 4-4, the slew rate checker can be initialized with a minimum slew rate to be respected and constant direction.

Figure 4-4 Slew Rate Checker Diagram



SVA Slew Rate Checker

You can specify an minimum slew rate with the `.slew` parameter and the direction with `.direction` parameter. This module is synchronous so you need to specify a clock with `clk` input and an analog node `vin` to be checked.

```
ams_sync_slew_checker #(.slew(sl),
                        .direction(dir))
   sync_rising_slew_chk(clk, vin);
```

Example 4-7 shows how to instantiate one `ams_async_slew_checker` modules to verify that `vin` slew rate remains higher that 0.5 V/ns.

Example 4-7 Checking Analog Node Slew Rate Using SVA Checker

```
`include "vmm_ams.sv"
module top;
  logic clk=1'b0;
  real vin;

  ams_sync_slew_checker #(.tolerance(0.5),
                          .direction(+1))
    sync_rising_slew_chk (.clk(clk),.vin(vin));

  always #10 clk = ~clk;

endmodule
```

VMM Slew Rate Checker

Similarly, VMM-AMS provides a slew rate checker that is modeled using `vmm_xactor`. The main difference is that it can be explicitly started/stopped and its parameters can be changed during simulation. It supports either synchronous or asynchronous sampling.

Example 4-8 shows how to instantiate two `vmm_ams_slew_checker` modules to verify that `ana_if.vin` signal slew rate remains greater than 0.5V/ns. It constantly checks the value of `ana_if.vin` which can be attached to an analog node or a real. The checking is performed synchronously, i.e. on each

positive edge of `clk`. A message based on `vmm_log` is emitted whenever the checker detects a violation. By default, a `vmm_error` is returned.

Note: In Example 4-8, the checker is explicitly started and stopped after 1000ns. Then, its output slew rate is changed to 0.45V/ns.

Example 4-8 Checking Analog Node Slew Rate Using VMM Checker

```
module top;
  logic clk=1'b0;
  ams_src_if ana_if(clk);
  vmm_ams_slew_rate_checker#(0.5,+1)
                    pos_chk = new("Chk", "", 0, ana_if);
  vmm_ams_slew_rate_checker#(0.5,-1)
                    neg_chk = new("Chk", "", 0, ana_if);
  always #10 clk = ~clk;

  initial begin
     pos_chk.start_xactor();
     neg_chk.start_xactor();

     #1_000 pos_chk.stop_xactor();
     pos_chk.set_params(0.45,0.05);
     pos_chk.start_xactor();

  end
endmodule
```

Frequency Checker

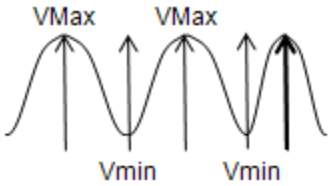
The frequency checker verifies that the input signal frequency remains within a specified reference with a +/- tolerance.

For efficiency reasons, the SVA implementation uses a periodic clock to sample the signal and a control signal that determines when the check must be performed. The VMM implementation if fully controllable and also supports synchronous/asynchronous data sampling.

This checker can be useful for verifying that some analog components, such as frequency generators, PLL, VCOs are able to drive an output with a stable frequency for different capacitance, processes, supply voltage, temperature, etc.

As shown in Figure 4-5, the frequency checker can be initialized with a reference frequency to be respected with a given tolerance. It basically measures when the voltage goes to Vmax, then Vmin to determine the half period. It again checks the next Vmax and Vmin to measure the frequency.

Figure 4-5 Frequency Checker Diagram



VMM Frequency Checker

VMM-AMS provides a frequency checker that is modeled using `vmm_xactor`. It can be explicitly started/stopped, its parameters can be changed during simulation. It supports either synchronous or asynchronous sampling.

Example 4-9 shows how to instantiate one `vmm_ams_frequency_checker` module to verify that `ana_if.vin` signal frequency remains equal to 2.5MHz +/- 1%.

The frequency is measured when `ana_if.vin` goes from -1.25V to 2.5V, then goes to -1.25V and returns to 2.5V.

It constantly checks the value of `ana_if.vin` which can be attached to an analog node or a real. The checking is performed synchronously, i.e. on each positive edge of `clk`. A message based on `vmm_log` is emitted whenever the checker detects a violation. By default a `vmm_error` is returned.

Note:In Example 4-9, the checker is explicitly started and stopped after 1000ns. Then, its frequency is kept to 2.5MHz but the voltage swing is changed to -450mV / +500mV.

Example 4-9 Checking Analog Node Slew Rate Using VMM Checker

```
module top;
  logic clk=1'b0;
```

```
  ams_src_if ana_if(clk);

  vmm_ams_frequency_checker#(-1.25,+2.5, 2.5E6)
          saw_freq_meas = new("SAWTOOTH FCHECK", "0", 0,
                              aif.sample);

  initial begin
     saw_freq_meas.start_xactor();
     saw_freq_meas.start_xactor();

     #1_000 saw_freq_meas.stop_xactor();
     saw_freq_meas.set_params(-0.45,0.5);
     saw_freq_meas.start_xactor();

  end
endmodule
```

5

AMS Source Generators

This chapter explains how to use VMM-AMS pre-defined voltage generators and also provides some guidelines for implementing the same.

VMM-AMS contains a few source generators that can be bound to internal or external nodes. These generators are responsible for driving a particular traffic to your 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, square are also available. Injection of random voltage is also possible.

Additionally, source generators can be combined together to inject more complicated traffic. A typical situation is to add white noise on top of a sine traffic. 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. In this case, the signal of interest can be represented by a small voltage fluctuation superimposed on a voltage carrier.

These source generators are available as standalone module and as VMM transactor. Both the implementations are written in SystemVerilog and are equivalent. The main difference is that the generators which are based on VMM transactor provide more flexibility in terms of reuse and controllability.

Table 5-1 provides a complete list of pre-defined voltage source generators which are written in SystemVerilog:.

Table 5-1 VMM-AMS Voltage Generators

Generic Voltage Source Generator	Provides base class infrastructure for building voltage generators
Sawtooth Source Generator	Base class aimed at generating sawtooth waveforms with minimum/maximum voltage and frequency
Sine Source Generator	Base class aimed at generating sine waveforms with minimum/ maximum voltage and frequency
Square Source Generator	Base class aimed at generating square waveforms with minimum/maximum voltage and frequency
Random Source Generator	Base class aimed at generating random waveforms with minimum/maximum random voltage sweep

VMM Real Support

As explained in Chapter 2, "Mixed-AMS Co-Simulation", the main communication media between the digital simulator and the analog simulator is achieved by converting SystemVerilog *real* from/to *electrical*. However, the SystemVerilog language as specified in the IEEE LRM 2008 does not support randomization or functional coverage of the *real* construct.

VMM-AMS provides a base class called `vmm_real` that allows to randomize *real* values. Additionally functional coverage groups can be written to take advantage of this base class, making it possible to combine the *real* construct with other parameters such as the configuration, testbench parameters, etc.

As shown in Example 5-1, the `vmm_real` is constructed with arguments allowing to assign it to the real value. Its default accuracy is needed to internally convert *real* to *integer* and *integer* to *real*. A `vmm_log` handle can be made available, which is useful for issuing unified messages.

Example 5-1 `vmm_real` Constructor

```
function vmm_real::new(real r=0.0,
                       real acc=`VMM_AMS_REAL_DEF_ACCURACY,
                       vmm_log log = null);
```

An important aspect of VMM-AMS is the possibility to randomize real values. Though this capability is not available in SystemVerilog, it is made possible with the `vmm_real::urandom()` method.

As shown in Example 5-2, a `vmm_real` instance is constructed with a default value of 0.0 and randomized with boundaries comprised between 0 to 1.8. By using this technique, the `vmm_real` instance can be used to deposit a voltage value to an analog node, which can take random values from 0.0 to 1.8V.

Note: You are using a default distribution that is similar to white noise injection.

Example 5-2 Randomizing vmm_real Instance

```
program automatic test;
  vmm_ams_real r  = new(0.0);

  initial begin
    int i;
    for(i=0; i<100; i++) begin
      r.urandom(0.0,1.8);
      $display("i=%0d - r=%f", i, r.get_real());
    end
  end
endprogram
```

Another important aspect of VMM-AMS is the possibility to automatically compare real values with a predefined tolerance. This is necessary for modeling reference models or to wait for an analog node the reach a given value. This capability is available with the `vmm_real::cmp(a,b,tol)` method.

As shown in Example 5-3, the `vmm_real` method can be used to wait for a particular analog node to reach a defined real value. In this example, the task `check_frequency` measures the frequency of the `v` analog node, which is in the `aif.sck` clocking block interface. This task waits for `v` to reach `vmax`, then `vmin` and measures `t0`. Similarly, it waits for `v` to reach again vmax, then vmin and measure `t1`. Therefore, the frequency can simply be calculated by subtracting `t0` to `t1`.

Example 5-3 Waiting for a vmm_real Value

```
task check_frequency(real frequency, tolerance=0.01);
  real t0,t1,f;

  wait(vmm_ams_real::cmp(aif.sck.v,vmax,tolerance));
  wait(vmm_ams_real::cmp(aif.sck.v,vmin,tolerance));
  t0=$realtime;

  wait(vmm_ams_real::cmp(aif.sck.v,vmax,tolerance));
  wait(vmm_ams_real::cmp(aif.sck.v,vmin,tolerance));
  t1=$realtime;

  f=1/(t1-t0);
  ...
```

In conjunction with Example 5-3, Example 5-4 shows how to use `vmm_real::cmp` method to compare the measured frequency against the expected one with a pre-defined tolerance of +/- 1%.

Example 5-4 Comparing to a vmm_real Value

```
  ...
  if (!(vmm_ams_real::cmp(f,frequency,tolerance)))
    `vmm_error(this.log,
       $psprintf("Expecting f=%f, %f with tol=%0.3f%%",
                 frequency, f, (tolerance*100)));
endtask
```

VMM Mathematical Functions

VMM-AMS provides a rich set of mathematical functions that are necessary for efficient modeling of transactors, traffic and reference models. All the available C functions that are implemented in the `<math.h>` library are exposed to the `vmm_ams_math` package.

Table 5-2 provides the complete list of available functions:

Table 5-2 Available Mathematical Function in `vmm_ams` Package

function real acosh(input real a)
function real acos(input real a)
function real asinh(input real a)
function real asin(input real a)
function real atan2(input real a, input real b)
function real atanh(input real a)
function real atan(input real a)
function real cbrt(input real a)
function real ceil(input real a)
function real cosh(input real a)
function real cos(input real a)
function real erfc(input real a)
function real erf(input real a)
function real exp(input real a)
function real expm1(input real a)
function real fabs(input real a)
function real floor(input real a)
function real fmod(input real a, input real b)
function real frexp(input real a, input integer b)
function real gamma(input real a)
function real hypot(input real a, input real b)
function int ilogb(input real a)

function int isnan(input real a)

function real j0(input real a)

function real j1(input real a)

function real jn(input int i, input real a)

function real ldexp(input real a, input integer b)

function real lgamma(input real a)

function real log10(input real a)

function real log1p(input real a)

function real logb(input real a)

function real log(input real a)

function real modf(input real a, input real b)

function real nextafter(input real a, input real b)

function real pow(input real a, input real b)

function real remainder(input real a, input real b)

function real rint(input real a)

function real scalb(input real a, input real b)

function real sinh(input real a)

function real sin(input real a)

function real sqrt(input real a)

function real tanh(input real a)

function real tan(input real a)

function real y0(input real a)

function real y1(input real a)

function real yn(input int i, input real a)

Note: These functions are automatically available by simply importing the `vmm_ams.sv` file in your SystemVerilog files.

Voltage Source Generators

Generic Source Voltage Generator

The `vmm_ams_src_gen` base class provides the foundation for building source generators such as, sawtooth, sine, square and random.

This section describes how to create your own source generator and also explains how to leverage from its infrastructure.

Note: You cannot use this base class as is because it is virtual and it should be extended.

The purpose of `vmm_ams_src_gen` generic source generator is to act as a container for defining minimum and maximum voltage boundaries, ease the conversion of integer to real. Also, the current voltage value can be posted to a channel. This helps in integrating the source generator with other VMM components such as, Multi-Stream scenario generators, transactors, etc.

Setting Minimum and Maximum Voltages

Minimum and maximum voltage boundaries are set while you construct the voltage generator. The constructor arguments are as follows:

```
vmm_ams_src_gen::new(
    real v_min=0.0,
    real v_max=1.0,
```

```
real accuracy=`VMM_AMS_REAL_DEF_ACCURACY,
vmm_log log = null);
```

Minimum and maximum voltage boundaries can be returned with `get_v_range()` method or set during simulation with the `set_v_range()` method.

```
set_v_range(real v_min, real v_max);
get_v_range(output real v_min, output real v_max);
```

Getting Voltage

The voltage generator returns to latest voltage with the `get_voltage()` method. This method must be overriden with your source generator implementation.

Setting Simulator Time Precision

As most of the source generators require to have access to the simulator time for calculating equations, it is sometimes required to set the simulator time precision. This is necessary when you use specific values for the ‘`timescale` directive.

Changing the source generator time precision is achieved by using the `set_time_precision()` method.

As shown in Example 5-5, the testbench ‘`timescale` is set to `1us/1ns`, which means the simulator time precision is now `1µs`. To assign a similar time precision to the `saw_gen` instance, the value 1.0E-6 is passed to the `set_time_precision` method.

Example 5-5 Changing Generator Time Precision

```
`timescale 1us/1ns

`include "vmm_ams.sv"
```

```
module top;
  logic sample=0;

  ams_src_if aif(sample);
  vmm_ams_sawtooth_voltage_gen saw_gen = new(-1.25,
                                             +1.25, 2.5E3);
  initial begin
    saw_gen.set_time_precision(1.0E-6);
```

Putting Voltage to a VMM Real Channel

The voltage generator can be used to return the current voltage by setting the `get_voltage()` to `vmm_ams_real channel`.

```
vmm_ams_src_gen::put(vmm_ams_real_channel chan);
```

Implementing Custom Voltage Source Generator

As explained in the previous section, the `vmm_ams_src_gen` base class can be extended to build your own source generator.

The required steps are,

- Extend the `vmm_ams_src_gen` base class.
- Override the `new` method with your new arguments and pass them to the parent base class. Store additional ones internally if needed.
- Override the `get_voltage` method with your custom source generator.

The next example shows how to model a RC low pass filter driven by a constant voltage. The equation for this source generator is,

$$V_{out} = (V_{max} - V_{min}) \times e^{-\frac{t}{RC}}$$

As shown in Example 5-6, the corresponding source code for this generator is,

Example 5-6 Modeling a Custom Generator

```
class rc_voltage_gen extends vmm_ams_src_gen;
  local real vdd;
  local real T;
  local real rc;

  function new(real v_min=0, real v_max=1.0,
               real f=1.0E6, real rc=1.0E-6);
    super.new(v_min, v_max);
    this.rc = rc * 1.0E9;
    this.set_frequency(f);
  endfunction

  virtual function real get_voltage();
    real r = vdd * exp(-$realtime/rc) +
             this.v_min.get_real();
    this.v.set_real(r);
    return r;
  endfunction

  virtual function void set_frequency(real f);
    this.T = (1.0E9 / f);
    this.vdd = this.v_max.get_real() -
               this.v_min.get_real();
  endfunction
endclass
```

Example 5-7 shows how to instantiate this custom source generator - with Vmin=-1V, VMax=1V, Freq=1MHz, RC=200ns - and assign its value to the `in` real node at regular times:

Example 5-7 Instantiating a Custom Generator

```
module top;
  real in;
  ...
  rc_voltage_gen rc_gen = new(-1.0, +1.0, 1.0E6, 2.0E-7);

  always @(posedge sample) begin
    in = rc_gen.get_voltage();
  end
  ...
endmodule
```

Sawtooth Voltage Generator

The `vmm_ams_sawtooth_voltage_gen` source generator provides a real value that can be used to drive analog voltage node with a sawtooth shape.

Example 5-8 shows how to set the minimum/maximum voltage and frequency with the constructor. Default values are 0.0 to 1.0V, with a frequency of 1MHz.

Example 5-8 Sawtooth Generator Constructor

```
class vmm_ams_sawtooth_voltage_gen extends
                                        vmm_ams_src_gen;
  ...
  function new(real v_min=0,
               real v_max=1.0,
               real f=1.0E6);
    ...
  endfunction
  ...
```

```
endclass
```

The equation of this generator is,

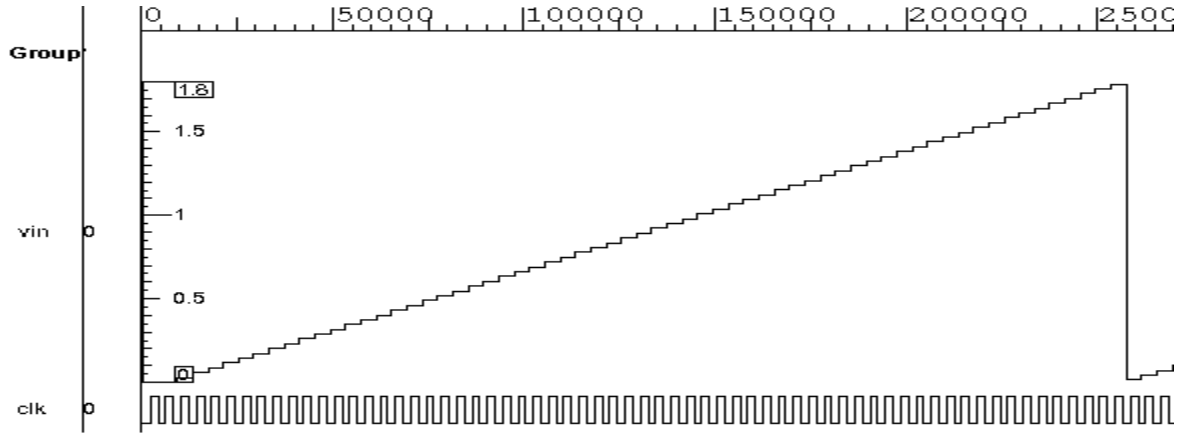
$$V_{out} = (V_{max} - V_{min}) \times mod\left(\frac{t}{T}, T\right) + V_{min}$$

Example 5-9 shows how to instantiate this sawtooth voltage generator - with Vmin=-1V, VMax=1V, Freq=3.84MHz - and assign its value to the `in` real node at regular times.

Example 5-9 Instantiating a Sawtooth Generator

```
module top;
  real in;
  ...
  vmm_ams_sawtooth_voltage_gen sawtooth =
     new(0.0, 1.8, 3.84E6); // 0 to 1.8V, 3.84MHz

  always @(posedge sample) begin
    in = sawtooth.get_voltage();
  end
  ...
endmodule
```



Sine Voltage Generator

The `vmm_ams_sine_voltage_gen` source generator provides a real value that can be used to drive analog voltage node with a sine shape.

Example 5-10 shows how to set the minimum/maximum voltage and frequency with the constructor. Default values are 0.0 to 1.0V, with a frequency of 1MHz.

Example 5-10 Sine Generator Constructor

```
class vmm_ams_sine_voltage_gen extends
                                        vmm_ams_src_gen;
  ...
  function new(real v_min=0,
               real v_max=1.0,
               real f=1.0E6);
    ...
  endfunction
  ...
endclass
```

The equation of this generator is,

$$V_{out} = (V_{max} - V_{min}) \times \sin(2\pi \times F \times t) + V_{min}$$

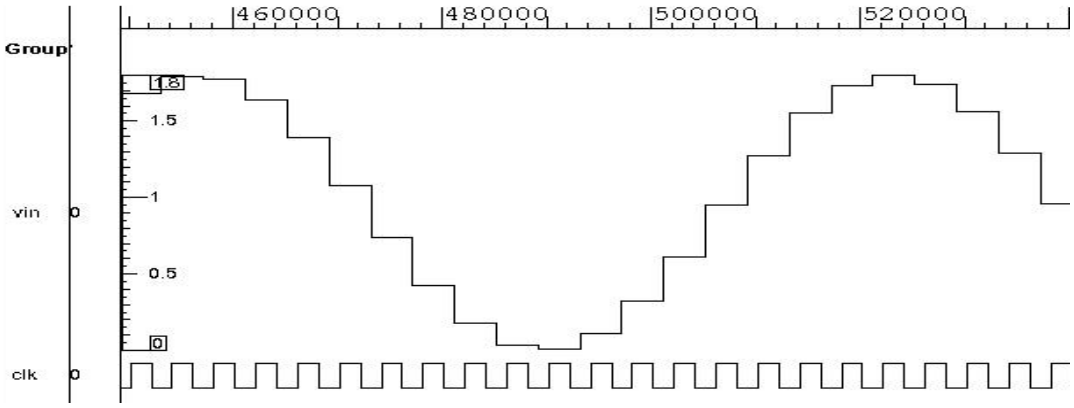
Example 5-11 shows how to instantiate this sine voltage generator - with Vmin=-1V, VMax=1V, Freq=4MHz - and assign its value to the `in` real node at regular times.

Example 5-11 Instantiating a Sine Generator

```
module top;
  real in;
  ...
```

```
  vmm_ams_sine_voltage_gen sine =
     new(0.0, 1.8, 4.0E6); // 0 to 1.8V, 4MHz

  always @(posedge sample) begin
    in = sine.get_voltage();
  end
  ...
endmodule
```



Square Voltage Generator

The `vmm_ams_square_voltage_gen` source generator provides a real value that can be used to drive analog voltage node with a square shape.

Example 5-12 shows how to set the minimum/maximum voltage, frequency and duty cycles with the constructor. Default values are 0.0 to 1.0V, with a frequency of 1MHz and a duty cycle of 50%.

Example 5-12 Square Generator Constructor

```
class vmm_ams_square_voltage_gen extends
                                    vmm_ams_src_gen;
  ...
  function new(real v_min=0,
```

```
                real v_max=1.0,
                real f=1.0E6,
                real duty=0.5);
    ...
  endfunction
  ...
endclass
```

Example 5-13 shows how to instantiate this square voltage generator - with Vmin=-1V, VMax=1V, Freq=10MHz, Duty cycle=25% - and assign its value to the `in` real node at regular times.

Example 5-13 Instantiating a Square Generator

```
module top;
  real in;
  ...
  vmm_ams_square_voltage_gen square =
    new(0.0, 1.8, 10.0E6,0.25); // 0 to 1.8V, 10MHz, 25%

  always @(posedge sample) begin
    in = square.get_voltage();
  end
  ...
endmodule
```

Random Voltage Generator

The `vmm_ams_random_voltage_gen` source generator provides a real value that can be used to drive analog voltage node with a random shape.

Example 5-14 shows how to set the minimum/maximum voltage with the constructor. Default values are 0.0 to 1.0V.

Example 5-14 Random Generator Constructor

```
class vmm_ams_random_voltage_gen extends
```

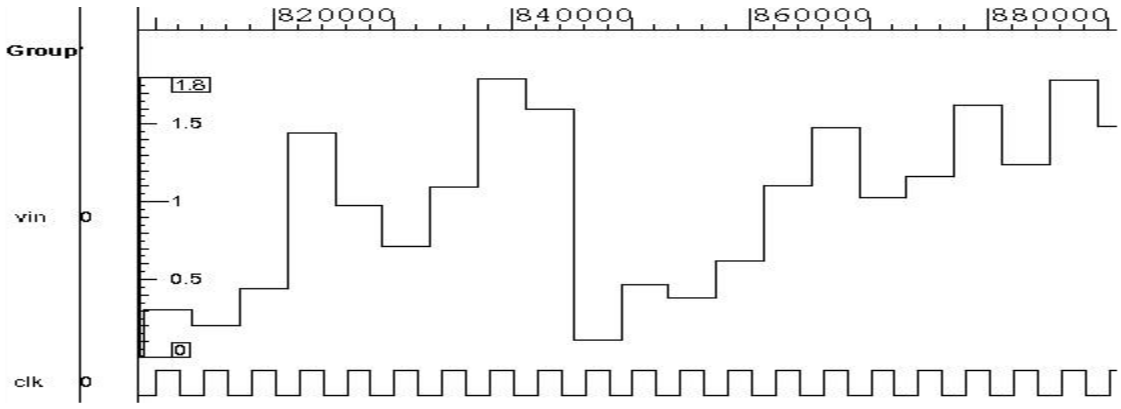
```
                                        vmm_ams_src_gen;
  ...
  function new(real v_min=0, real v_max=1.0);
    ...
  endfunction
  ...
endclass
```

Example 5-15 shows how to instantiate this sawtooth voltage generator - with Vmin=-1V, VMax=1V, Freq=10MHz - and assign its value to the `in` real node at regular times.

Example 5-15 Instantiating a Random Generator

```
module top;
  real in;
  ...
  vmm_ams_random_voltage_gen random =
     new(-0.05, +0.05); // -50mV to +50mV

  always @(posedge sample) begin
    in = random.get_voltage();
  end
  ...
endmodule
```



Easy Integration With VMM Source Generators

As described in the previous sections, all 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 the pre-defined SystemVerilog interface and better controllability (therefore, reuse at SoC). Additionally, this component comes with pre-defined 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 reaches the given state.

As shown in Example 5-16, the `sine_gen` generator is used to drive a sine waveform to the `amp` analog IP. The connection is established with the `aif` interface. The `src_gen` transactor is used to register the `sine_gen` generator and to drive the `aif.sck.v` analog node on each positive edge of `sample`.

Example 5-16 VMM Source Generator Associated With Sine Generator

```
module top;
  real out;
  logic sample=1;

  ams_src_if aif(sample);
  amp#(1.00) dut(aif.sck.v, out);
  vmm_ams_sine_voltage_gen sine_gen = new(-1.0, +1.0,
                                          1.0E6);
  ams_src_xactor src_gen = new("SINE", "0", 0,
                               sine_gen, aif.drive);
```

```
...
```

As shown in Example 5-17, the `src_gen` transactor can be started and stopped on purpose.

Note: This transactor gets notified whenever its embedded `sine_gen` generator reaches its half period, i.e. π.

By using this approach, it's possible to ensure the amplifier returns to 0.0V at regular times.

Example 5-17 VMM Source Generator Associated With Sine Generator

```
...
initial begin
  src_gen.start_xactor();
  src_gen.wait_for_half_period();
  if(!vmm_ams_real::cmp(out, 0.0))
     `vmm_error(src_gen.log,
          $psprintf("Expecting PI = 0.0V - got %f",
                                              out));
```