Addressing SOC/IP Verification Framework Creation with UVM Centric Mechanisms

Adiel Khan, Amit Sharma

Synopsys
Agenda

• IP SOC Verification Tests/Tasks
• Quick UVM Introduction and Typical Usage scenarios
• Using UVM for mixed-AMS SOC verification
• Leveraging UVM TLM for System Level Verification
• Unified Register modeling between C and SV
• Enabling an UVM Testbench for simulation acceleration
# Verification & Test Tasks

<table>
<thead>
<tr>
<th>Verification Tasks</th>
<th>Block Sim</th>
<th>SoC Sim</th>
<th>Gate Sim</th>
<th>Emulation</th>
<th>Silicon</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connectivity</strong></td>
<td>CRV Interface Protocol Compliance</td>
<td>SW Register Sweep Reset Registers</td>
<td>SW Limited Reg Sweep Reset Some Registers</td>
<td>Full Register Sweep Multiple Configs</td>
<td>Full Register Sweep Multiple</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td>CRV Block Level Functional Bugs</td>
<td>BFM CRV SW Simple Block to Block</td>
<td>SW Limited Functionality Minimal Boot Code</td>
<td>SW Driver Functionality Bring Up Tests</td>
<td>C/SWSilicon Bringup Bring Up Tests</td>
</tr>
<tr>
<td><strong>Performance</strong></td>
<td>CRV Perf SW Perf Back to back frames</td>
<td>SW Bare Metal OS Block to Block</td>
<td>None</td>
<td>Performance Test Verify Peak Performance</td>
<td>Boot OS Applications Performance Stress Tests</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>BFM C/VPI tests Initialize + Basic Driver</td>
<td>BFM + VPI/C C/Bare OS Initialize + Basic Driver</td>
<td>SW Limited Functionality Initialize + Small Test</td>
<td>Boot OS Applications Real Software</td>
<td>Boot OS Applications Real Software</td>
</tr>
</tbody>
</table>
HW Based & SW Based Methodology

Software Team & Skills move down towards hardware

- Software Development
- Software Drivers
- Application Software

- SoC driver testing
- Bare Metal OS
- Minimal Real OS

- RTL Block Verification
- SoC Connectivity
- SoC Block Traffic

Virtual Platform
- Hardware Prototype
- Silicon

- Emulation
- Test runs on embedded CPU

- RTL Simulation
- Test runs on embedded RTL CPU

- RTL Simulation
- CPU is Bus BFM
- UVM Methodology

Hardware Team & Skills move up towards software

Mix depends on team skills

Software Development

Hardware Verification

Software Team & Skills move down towards hardware

Hardware Team & Skills move up towards software
Complete UVM Ecosystem

Coverage & Analysis

Verification Planning

Reg Model generators

Constraint Solver

Template Generators

Protocol Debug

Native UVM VIP

UVM-Aware Debug

UVM AMS Testbench

UVM TLM/SC Adaptors

Constraint Solver
Verification Goal

- Ensure full conformance with specification:
  - Must avoid **false** passes

### Testbench Simulation result
- **Pass**
- **Fail**

### RTL code
- **Good**
- **Bad (bug)**

- **Tape out!**
- **Debug testbench**
- **Debug RTL code**

**How do we achieve this goal?**

False pass results in shipping a bad design
Coverage-Driven Verification

- Focus on uncovered areas
- Trade-off authoring time for run-time
- Progress measured using functional coverage metrics
Phases of Verification

Start with fully random environment. Continue with more and more focused guided tests

- Preliminary Verification
- Broad-Spectrum Verification
- Corner-case Verification

Goal

% Coverage

Time

Difficult to reach Corner-case Verification

Build verification environment
Typical Testbench Architecture

- SystemVerilog testbench structure
UVM Testbench Architecture

Test Entry

Test Layer

Container Layer

Transaction Layer

Physical Layer

Program

Test

Environment

Master Agent

Slave Agent

Sequencer

Coverage

Scoreboard

Driver

Monitor

DUT

Harness Module

reg

interface

reg

DUT

reg

interface
UVM Encourages Encapsulate for Reuse

- Structure should be architected for reuse

Test instantiates the environment and modifies the environment on a testcase by testcase basis.

Agents, coverage and scoreboard should be encapsulated in an environment.

Sequencer, driver and monitor associated with an interface should be encapsulated as an agent for that interface.
UVM Structure is Scalable

- Agents are the building blocks across test/projects
Structural Support in UVM

- Structural & Behavioral
  - uvm_component
    - uvm_test
    - uvm_env
    - uvm_agent
    - uvm_sequencer
    - uvm_driver
    - uvm_monitor
    - uvm_scoreboard

- Communication
  - uvm_*_port
  - uvm_*_socket

- Data
  - uvm_sequence_item
Typical UVM Usage scenarios

• UVM based Constrained Random Verification (CRV) for RTL Block/IP Verification using
  ✓ UVM based VIPs with Constrained Random tests
  ✓ Usage of standard 3\textsuperscript{rd} Party UVM VIPs in IP/Subsystem Verification

• Generating SoC Block Traffic

• IP verification using Re-use of IP level testbench at SOC/System level
  – UVM VIPs as peripherals
  – UVM VIPs as passive monitors
  – Reuse of UVM Register layer from block to SOC level
UVM Coverage Driven Verification

Testcase A

Sequencer

Driver

Monitor

Driver

Monitor

Driver

Monitor

System-level transactions

Low-level transactions

Block-level transactions

Block to System Sequences/Cover Group reuse

Refer to existing coverage object no need to do rewrites of coverpoints or covergroups

DUT

Functional Coverage Model
**UVM Constrained Random Verification**

Quick tracing of sequences through phases and sequencers

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Sequence ID</th>
<th>Start</th>
<th>Finish</th>
<th>Phase</th>
<th>Thread</th>
<th>Sequencer</th>
<th>Sequencer ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop_read_modify_write_seq</td>
<td>loop_read_modify_write_seq@1</td>
<td>0</td>
<td></td>
<td>run(common)</td>
<td>96</td>
<td>...masters[0].sequencer xbus_master_sequencer@1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read_modify_write_seq@1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read_byte_seq@1</td>
<td>0</td>
<td>110</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@12</td>
<td>60</td>
<td>110</td>
<td></td>
<td>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write_byte_seq@1</td>
<td>110</td>
<td>210</td>
<td></td>
<td>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@16</td>
<td>120</td>
<td>210</td>
<td></td>
<td>240</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read_byte_seq@3</td>
<td>210</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@20</td>
<td>220</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read_read_modify_write_seq</td>
<td>read_read_modify_write_seq@2</td>
<td>0</td>
<td></td>
<td>run(common)</td>
<td>116</td>
<td>...masters[1].sequencer xbus_master_sequencer@2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read_modify_write_seq@2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read_byte_seq@2</td>
<td>0</td>
<td>160</td>
<td></td>
<td>241</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@13</td>
<td>60</td>
<td>160</td>
<td></td>
<td>243</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>write_byte_seq@2</td>
<td>160</td>
<td>260</td>
<td></td>
<td>241</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@18</td>
<td>170</td>
<td>260</td>
<td></td>
<td>241</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>read_byte_seq@4</td>
<td>260</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@22</td>
<td>270</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>master_memory_seq</td>
<td>master_memory_seq@1</td>
<td>0</td>
<td></td>
<td>run(common)</td>
<td>138</td>
<td>...s0.slaves[0].sequencer xbus_slave_sequencer@1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>master_memory_seq@2</td>
<td>0</td>
<td></td>
<td>run(common)</td>
<td>150</td>
<td>...s0.slaves[1].sequencer xbus_slave_sequencer@2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@9</td>
<td>0</td>
<td></td>
<td></td>
<td>227</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>master_memory_seq@3</td>
<td>0</td>
<td></td>
<td>run(common)</td>
<td>192</td>
<td>...s0.slaves[2].sequencer xbus_slave_sequencer@3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@10</td>
<td>0</td>
<td></td>
<td></td>
<td>231</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>master_memory_seq@4</td>
<td>0</td>
<td></td>
<td>run(common)</td>
<td>204</td>
<td>...s0.slaves[3].sequencer xbus_slave_sequencer@4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>xbus_transfer@11</td>
<td>0</td>
<td></td>
<td></td>
<td>235</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Enables Next-Generation VIP Architecture

Customization
- Sequence Collection
- Configuration Creator

Coverage
- User Verification Plan
- Protocol Test Plan
- Coverage Database

Testbench
- Native SystemVerilog
  - User Tests
  - Test Suite
- Native UVM

Debug
- Protocol Analyzer
- VIP Source Visibility
- DVE

Test Suite
- Time to 1st Test
  - Configuration Creation GUI
  - Built-in test plan
  - Sequence Collection

Time to Verify
- Native SystemVerilog
- No wrappers
- Up to 4x faster

Time to Debug
- Protocol-aware debug
- Source code visibility
- Error Diagnostics

Time to Coverage
- Built-in coverage
- Verification Plan
- Test Suite
- Sequence Library
Performance Validation for AXI Protocol Using UVM testbench

Measure bus bandwidth based on traffic profiles

Performance Requirements

Specification

Final Silicon and Applications

System Architects

Verification Engineers

Performance Constraints

<table>
<thead>
<tr>
<th>Setting</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write transaction latency</td>
<td>Min, Max, Avg latency</td>
</tr>
<tr>
<td>Read transaction latency</td>
<td>Min, Max, Avg latency</td>
</tr>
<tr>
<td>Write transaction throughput</td>
<td>Max, Min Number of Bytes/Time Unit</td>
</tr>
<tr>
<td>Read transaction throughput</td>
<td>Max, Min Number of Bytes/Time Unit</td>
</tr>
</tbody>
</table>
Performance Analysis

- User sets
  - Performance constraints
  - Recording interval
- Registration using UVM callbacks
- UVM_ERROR for violation
- Dynamically Configurable
- Prints Performance Summary

*******************************************************************************
PERFORMANCE REPORT FOR MASTER 0:
*******************************************************************************

-----------------------------------------------------------------------------
Interval Start Time: 0.000000; Interval End Time: 5425.000000
Configured avg_max read xact latency: 15000.000000;
Observed avg_max read xact latency: 360.000000
-----------------------------------------------------------------------------
Metric Compliance Checking

- Visualize Performance Data in Graphs/charts/spreadsheets
- Performance constraints can be fed into spreadsheets/EDA tools for checking performance data

TCL File: TCL

CSV File: CSV

Constraint Editor: Apply
Architecture Exploration, TLM Verification, leveraging System C reference models

LEVERAGING UVM TLM FOR SYSTEM LEVEL VERIFICATION
Mixed language/abstraction level Challenge

- Single, golden testbench for TL and RTL
- Transaction-level models in SystemC or SystemVerilog
- Verification in a mixed language with different abstraction levels
What is OSCI TLM?

• TLM 2.0
  – Standard for interoperability between memory mapped Bus model
  – Loosely timed and Approximately timed model
  – Used for Virtual platform, Architectural analysis, Golden model for Hardware verification
  – Provided Sockets(Transport, DMI and Debug interfaces)
  – Generic payload and extension mechanism
• TLM 1.0 is included inside TLM 2.0 standard (put, get, nb_put, nb_get, transport).
Reasons for Using TLM

- **Firmware / software**
  - Software development
  - Accelerates product release schedule
  - Available before RTL

- **Fast enough**
  - TLM
    - Ready before RTL

- **RTL**
  - Architectural modeling
  - Hardware verification

- **Test bench**
  - TLM = golden model

Represents key architectural components of hardware platform
Simulates much faster than RTL
TLM-2.0 across SV and SC

• Tool-specific mechanism
  – Not part of UVM
  – VCS: TLI-4
  – Open Source: UVM-Connect

SystemVerilog

Copy across at method call and return

SystemC
Methodology Overview: How does it work

Also Work from SC to UVM direction

Data Pkt is user-defined/TLM2.0 Generic Payload

TLI Adaptor

TLI bind function for connecting the SC initiator/target with TLI adaptor initiator/socket

TLI bind function connects the env.master.channel/env.master.socket to TLI adaptor Channel/Socket

Type converter

SystemC model

Ref model with OSCI TLM 2.0 Interface

Sequence

Function

UVM Testbench Architecture

Pkt

Data

PIN IF

SC

SV

SV-SC TLI

Channel

SC-SV TLI
Leveraging TLM2.0 for HW/SW Integration

**Early Test Bench Development**

1. Develop test bench infrastructure
2. Develop early test cases and scenarios

**HW/SW Co-Verification**

1. SW Driven Verification
2. SoC HW/SW integration
Virtual Platforms and TLM-2.0

- Virtual Platform written in SystemC
- Bus-based SoC
- TLM-2.0 models bus operations
  - Independent of bus protocol
# RTL & Fast Models

- **Speed and accuracy tradeoff**
  - RTL – full cycle accuracy
  - TLM – transaction accuracy
  - Some intermediate points also available (consider ROI)

<table>
<thead>
<tr>
<th>Model</th>
<th>TLM Virtualizer</th>
<th>TLM Co-Sim</th>
<th>RTL Co-Sim</th>
<th>RTL Backdoor</th>
<th>RTL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>Fast</td>
<td>Fast</td>
<td>Slow</td>
<td>Slow/Med</td>
<td>Slow</td>
</tr>
<tr>
<td>Accuracy</td>
<td>TLM</td>
<td>TLM</td>
<td>TLM</td>
<td>TLM</td>
<td>Cycle</td>
</tr>
</tbody>
</table>

Faster Speed (more TLM)  
Accuracy (More RTL)

Need to consider ROI for each of these environments
RTL in Virtual Platforms

ISS

Interconnect

VIP

A

B

C
Co-Simulation: Virtualizer TLM & RTL

TLM/RTL Co-Simulation
• Transaction level accuracy
• Fast until RTL is touched
• Minimal RTL to maintain speed
• Minimal design changes, only when RTL is inserted
Co-Simulation: TLM & RTL

TLM/RTL Co-Simulation
- Transaction level accuracy
- Mostly RTL
- Simulation performance similar magnitude range as RTL simulation speed,
- Design change required to add Pin Interfaces/Adapters
Protocol-Agnostic Tests

- Test written using Generic Payload
- Can work on any bus
VIP ARCHITECTURE FOR TLM GP
GP-Based Sequences in SV

GP Tests

GP Sequences

GP Sequencer

SNOOP Sequencer

GP-to-AMBA Layering

Driver

uvm_tlm_generic_payload + svt_amba_pv_extension

AMBA Transactions

Snoop request reactive sequence

GP to AMBA layering sequence

AMBA Transactions

GP Tests

GP Sequences

GP Sequencer

SNOOP Sequencer

GP-to-AMBA Layering

Driver

uvm_tlm_generic_payload + svt_amba_pv_extension

AMBA Transactions

Snoop request reactive sequence

GP to AMBA layering sequence
SystemC-Based Tests

Directed GP Sequence → GP Sequencer

SNOOP Sequencer

GP-to-AMBA Layering → Driver

Snoop request reactive sequence

GP to AMBA layering sequence

To FastModel

b_snoop

b_fwd

uvm_tlm_generic_payload + svt_amba_pv_extension

AMBA Transactions

SVT_AMBA_PV_EXTENSION
FastModel Connection

ARM FastModel

AMBA-PV Bridge

Code Memory

AMBA-PV Slave

UVMeconnect / TL-4

b_snoop

b_fwd

SNOOP Sequencer

AXI-to-GP Layering

One GP Sequence

GP Sequencer

SystemC

SystemVerilog
UVM Transaction Debug

New Transaction Based Debug Apps

Verification Platform

FSDB Trans, Attributes, Relations

Test-bench
VIPs
User Code
SoC/Processor

Library-Embedded Recording Interface
VIP-Embedded Recording Interface
PLI task-based API
C function API
Transaction Debug Apps

Enhancing Post Process Testbench Debug
Transaction Analyzer

Analysis and data mining tool for abstract data
Unified register Modeling between C and SV
Unified Register Model for C and SV

Challenges and Motivation

- To enhance register model interaction between C and UVM, a new mechanism is needed such that an identical and consistent register model can be accessed from either side.
- This framework can alleviate the need to maintain and verify two separate and disjointed models.
- Updates to this common register model from one side need to be immediately visible to the other and vice versa.

![Diagram of Unified Register Model for C and SV]

SV Testbench

- Stimulus/Sequences
- Unified UVM+C Register Model
- Compare Output

C++ Testbench

Accellera Systems Initiative

DVCon 2015

Design and Verification Conference and Exhibition

India
Implementing a Unified Model

- The standard UVM RAL (Register Abstraction Layer) models the registers and memories of the Design Under Test (DUT) in System Verilog (SV).

- A C-model of the design exists and has a C++ based environment. The APIs provided allow access of the register fields and memories in the SV-RAL model from the application level C code.

- The RAL-SV model can be updated from the C side or the SV testbench. Any updates in the RAL-SV model from the SV side are reflected in the C side.
Introduction to RAL-C++ interface

• Allows firmware and application-level code to be developed and debugged in VCS
• Preserve abstraction offered by the RAL
  – Hide physical address, field positions
• Provides C++ API to access RAL components
  – Fields, Registers
• Two versions of the RAL C++ API can be generated
  – Interface to RAL model using DPI
  – Stand-alone C++ code targeted to S/W
RAL C API

Application level code can now be verified against a simulation and then used, unmodified, in the final application
Execution timeline

- Execution non concurrent unlike code
- Rest of Simulation frozen when ‘C’ code is running
- Entire execution in ‘C’ is in ‘0’ time in the simulation timeline
- ‘Polling’ strategy can impact overall performance
- Interrupt driven service strategy more ideal

![Diagram showing execution timeline with 'C Code' and 'Simulation' with 'read' and 'write' operations.]
Using UVM for mixed-AMS SOC verification
Need for a UVM-AMS Testbench?

• No clear methodology for mixed-AMS SoC verification
  – Need interaction between all IPs, including AMS
  – Need signal access between analog IPs and others (xmrs/oomrs)

• Holes in mixed-AMS Block-level verification
  – Synchronous verification is not enough, Ref models, Flow automation needed for characterization, regression

• The solution for self-checking verification environment
UVM-AMS Testbench Overview

Technology for mixed-signal SoC functional verification

Basic Usage
- Electrical ↔ Real conversion
- Asynchronous analog events
- AMS toggle coverage

Intermediate usage
- AMS SystemVerilog assertions
- AMS SystemVerilog testbench
- AMS Checker Library
- SystemVerilog Real Number Modeling

Advanced usage
- UVM AMS testbench
- AMS Source generators
Real ↔ Analog Conversion

Easy XMR read access to internal analog signal voltage and current

\$snps_get_volt(anode)
\$snps_get_port_current(anode)

*anode*: full hierarchical analog node name

**Example:**

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

Easy XMR write access to internal analog signal voltage.

\$snps_force_volt(anode,val|real)
\$snps_release_volt(anode)

*anode*: full hierarchical analog node name
*val|real*: absolute value or real variable

**Example:**

```vHONE
real r;
initial begin
    $snps_force_volt(top.i1.ctl,0.0);
end
always @(posedge clk) begin
    r <= r + 0.1;
    $snps_force_volt(top.i1.ctl,r);
end
```

**accellera**
Logic ↔ Analog Conversion

Automatic insertion of a2d connect models between SystemVerilog and SPICE. User can re-define the threshold

Example:
assign verilog_wire =
    top.i1.i2.x1.clk;
initial begin
    verilog_reg =
        top.i1.i2.x1.strb;
    ...
Asynchronous Analog Events

Example:
always @(snps_cross($snps_get_volt(top.i1.ctl)-0.6,1))
begin
    $display("Signal ctl is raising above 0.6V");
end

Example:
always @(snps_above($snps_get_volt(top.i1.ctl)-0.6))
begin
    $display("Signal ctl is above 0.6V");
end
Instantiation of Analog DUT & Testbench

- DUT and testbench are instantiated and connected using SV interface
- VCS automatically inserts necessary e2r and r2e models

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

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

ana_test tst(ana_if);

always #10 clk = ~clk;
```
AMS Testbench Generators

UVM

class my_env extends uvm_component;
...
sv_ams_sine_voltage_gen#(-1.0, +1.0, 1.0E6) sGen_IN;

function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  uvm_resource_db#(virtual ams_src_if)::
    set("*", "uvm_ams_src_if", aif, this);
  sGen_IN = sv_ams_sine_voltage_gen#
    (-1.0, +1.0, 1.0E6)::type_id::create("sine", this);
endfunction

Sine Voltage Gen
- Vmax=1.0V,
- Vmin=-1.0V
- F=1.0MHz

Construct sin Wave generator. Default is auto-run throughout run_phase()
Immediate Assertions

Asynchronous

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

Asynchronous immediate assertion of analog node

- `top.ev`
  - `VDD=1.8V`
  - `0.6V`

- `top.vref.analog_node`

**Node is greater than VDD**
module test;
  ana_vref_if ana_if();
  spice_clk_ref ckref(ana_if.clk_out);

  sv_ams_frequency_checker#(-2.25, +2.25, 2.5E6) freq_meas = new("Freq Meas", "0", 0, ana_if.clk_out);

initial begin
  @(posedge rst); // Wait end of reset
  saw_freq_meas.start_xactor();
  #2000 saw_freq_meas.stop_xactor();
  saw_freq_meas.set_params(-1.8, +1.8, 2.5E6);
  saw_freq_meas.start_xactor();
endmodule
# AMS Testbench Checkers

<table>
<thead>
<tr>
<th>Checkers</th>
<th>Description</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sv_ams_threshold_checker</code></td>
<td>Checks that analog signal remains within a given high and low threshold.</td>
<td>![Graph](High and Low Threshold)</td>
</tr>
<tr>
<td></td>
<td>Can perform this check synchronously or asynchronously</td>
<td></td>
</tr>
<tr>
<td><code>sv_ams_stability_checker</code></td>
<td>Checks that analog signal remains below or above a given threshold.</td>
<td><img src="Stability" alt="Graph" /></td>
</tr>
<tr>
<td></td>
<td>Can perform this check synchronously or asynchronously</td>
<td></td>
</tr>
<tr>
<td><code>sv_ams_slew_checker</code></td>
<td>Checks that analog signal rises/falls with a given slew rate(+-tolerance).</td>
<td>![Graph](Slew Rate)</td>
</tr>
<tr>
<td></td>
<td>Can perform this check synchronously or asynchronously</td>
<td></td>
</tr>
<tr>
<td><code>sv_ams_frequency_checker</code></td>
<td>Checks that analog signal frequency is within a given tolerance</td>
<td><img src="Frequency" alt="Graph" /></td>
</tr>
</tbody>
</table>
UVM AMS Verification of A+D SoC

AMS TB contain analog blocks

DUT

CHECK

TEST

Coverage

Sine Voltage Gen
- Vmax=1.0V,
- Vmin=-1.0V
- F=1.0MHz

Construct sin Wave generator. Default is auto-run throughout run_phase()

class my_env extends uvm_component;
...
sv_ams_sine_voltage_gen#(-1.0, +1.0, 1.0E6) sGen_IN;

function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_resource_db#(virtual ams_src_if)::
        set("*", "uvm_ams_src_if", aif, this);
    sGen_IN = sv_ams_sine_voltage_gen#
        (-1.0, +1.0, 1.0E6)::type_id::create("sine", this);
endfunction
Other use: Verifying Analog IP before SoC integration

sv_ams_real

sv_ams_voltage components
Enabling an UVM Testbench for simulation acceleration
Typical UVM Testbench Environment

- Stimulus Generation: Constrained Random, Object-Oriented
- Testcase: Simulation Control
- Bus Functional Models (BFM’s)

- Responder, Slave (Memory)
- Result Analyzers
- Coverage Driven Verification
- Scoreboards
Environment Transformations for TBA

Two-Top Entity Approach

Partitioning the Active & Passive BFM’s
Complete UVM Ecosystem

- Coverage & Analysis
- Verification Planning
- Reg Model generators
- Constraint Solver
- Protocol Debug
- Native UVM VIP
- UVM-Aware Debug
- UVM AMS Testbench
- UVM TLM/SC Adaptors
- Template Generators
Questions