Verification Futures Conference

Austin Marriott South

Thursday 14 September 2023

Sponsored by

Participating Companies

Media Sponsor
Engineering Tomorrow’s Solutions
Your Capable Partner Today

- Robust Infrastructure
  Test Floor | Reliability & STPI Smart Lab
- Expert Engineering Team
  90% Technical Staff
- Quality Processes
  Upto 30% Cycle Time Reduction

CoE Technologies
- ARM, RISC-V Subsystem and Analog Block Development
- 5G and High Frequency RF Solutions
- High Performance Compute Solution
- Automotive Compliance Solution

Let’s Talk About Engineering New Possibilities

Automotive | Data Centre/Enterprise | Industrial/IoT | Avionics | Semiconductor

www.tessolve.com | Email: sales@tessolve.com
Agenda (AM)

08:30  Arrival: Breakfast and Networking
09:25  Welcome: Mike Bartley, Tessolve Semiconductor Ltd
09:30  **Keynote Speakers**
        Vivek Vedula, ARM Ltd
10:15  User Top Verification Challenges
        10:15  Alex Duhovich, Ericsson
10:30  Bahadir Erimli, Cadence Design Systems
11:00  Refreshments and Networking
11:30  **Multi-Track Session**
        **Track 1 - User Presentations on Formal Verification  [Lonestar Ballroom – Salon A+B]**
        11:30  Mike Bartley, Tessolve Semiconductor Ltd
        11:40  Divyang Agrawal, Tenstorrent, Inc
        12:10  Suneil Mohan, Intel Corporation
        **Track 2 - Training Session 1  [Lonestar Ballroom – Salon C]**
        11:30  Doug Smith, Doulos
        **Track 3 - UVM for AMS Verification  [Lonestar Ballroom – Salon D]**
        11:30  Peter Grove & Steven Holloway, Renesas  *Remote Presentation*
12:30  Lunch and Networking
Agenda (PM)

13:30  Larry Lapides (Imperas Software Ltd.)
14:00  Adnan Hamid (Breker Verification Systems)
14:20  Balram Naik Meghavath (Broadcom Ltd)
14:40  Hemendra Talesara (Bitstar Technologies)

15:00  Refreshments and Networking
15:30  Multi-Track Session

**Track 1 – Latest Topics in Verification  [Lonestar Ballroom – Salon A+B]**
15:30  Aditya Devarakonda, NXP Semiconductor
15:50  Bill Tiffany, SigmaSense LLC
16:10  Benjamin Delsol, UVMGEN

**Track 2 - Training Session 2  [Lonestar Ballroom – Salon C]**
15:30  Doug Smith, Doulos

**Track 3 – VHDL Verification  [Lonestar Ballroom – Salon D]**
15:30  Jim Lewis, SynthWorks Design Inc

16:30  Event Closes
Floor Plan
Notes
Mike Bartley
Tessolve Semiconductor Ltd
Senior Vice President – VLSI Design

Welcome Message

Biography

Dr Mike Bartley has over 30 years of experience in software testing and hardware verification. He has built and managed state-of-the-art test and verification teams inside a number of companies (including STMicroelectronics, Infineon, Panasonic and start-up ClearSpeed) and also advised a number of companies on organisational verification strategies (ARM, NXP and multiple start-ups).

Mike successfully founded and grew a software test and hardware verification services company to 450+ engineers globally, delivering services and solutions to over 50+ clients in a wide range of technologies and industries. The company was acquired by Tessolve semiconductors, a global company with 3000+ employees supporting clients in VLSI, silicon test and qualification, PCB and embedded product development in multiple vertical industries.

Mike has a PhD in Mathematics (Bristol University), and 8 MSc’s in various subjects including management, software engineering, computer security robotics and AI. He is currently studying (remotely) for an MSc in Blockchain and Digital Currency at the University of Nicosia, Cyprus

Tessolve would like to thank the sponsors and participants of the 2023 Verification Futures Conference
Notes
Vivek Vedula

Arm Ltd
Technical Director

Safety and Security challenges in hardware IP development

Keynote Speaker

Abstract
Ensuring the trustworthiness in computing is increasingly becoming a challenge in the interconnected world relying on electronic systems. Security and safety provide assurance that these systems are resilient to malicious attacks and malfunctioning components, respectively. Given the diverse and rapidly evolving market demands, the requirements for both new features and performance significantly increases the probability of security- and safety-related design flaws to remain undetected. This talk will describe the challenges during IP development in efficient identification of relevant risks, and their effective mitigation for safe and secure computing.

Biography
Vivek Vedula leads the SDL methodology architecture and development for hardware IPs at Arm. Prior to this, he held several roles at Intel, NXP and Oracle Labs spanning the areas of formal verification, post-silicon validation and HW-SW co-verification. Vivek holds a PhD degree in Electrical and Computer Engineering from the University of Texas at Austin.

Slides will be shared during presentation.
Alex Duhovich
Ericsson
PEU Silicon – IP Verification Methodology Lead

Ericsson’s Challenges of IP Development and Verification for Products with a Long Shelf Life

*Challenge Paper*

**Abstract**
Ericsson develops ASICs for radios which have a long shelf life and an even longer life cycle. It’s hard to have IP roadmaps with on-time requirements to allow for IP-centric planning and execution. This presentation will outline some of the challenges Ericsson IP teams have been facing in their quest to IP driven in a product driven market.

**Biography**
- 20+ years of ASIC/SoC design and verification experience
- BSEE from Drexel University, MEEE from University of Maryland College Park
- Most of career spent in the telecommunications industry
- Started at Ericsson in 2017
- Methodology lead since 2021
Notes
About myself
• 20+ Years, mainly in Telecommunications Industry (Hughes, Ericsson)
• Bachelor’s in EE from Drexel in 2000
• Master’s in EE from University of Maryland College Park in 2015
• At Ericsson since 2017
• Started in IP verification -> Team Lead -> Verification Methodology Lead
• Email: alexei.duhovich@ericsson.com
• LinkedIn: https://www.linkedin.com/in/aduhovich/

Ericsson at a glance:
A world leader in ICT and 5G
Purpose:
• To create connections that make the unimaginable possible
Vision:
• A world where limitless connectivity improves lives, redefines business, and pioneers a sustainable future
History:
• 140+ years of delivering groundbreaking solutions and innovative technology for good
Leader in Technology:
• Leading provider of Information and Communication Technology (ICT) to service providers
• 227.2 b. SEK (~ $27b) in Sales
• 54,000 patents

Solutions take many form factors
**Challenges**

- Product-centric development
- Long product shelf lives lead to requirements creep
- Requirement quality gaps lead to planning challenges and schedule slips
- General increasing complexity challenge with verification: Methodology scaling, Power, Security

**Product Centric Development**

- IP requirements based on product
- Requirements come in at the start of a project
- Conflicting requirements possible between projects
- Team to close all milestones for each project
- Difficult to maintain code base and version control
- Difficult to deliver to multiple projects at once

**Long Product Shelf Lives**

- Product lives of ~10 years
- Customer expects longevity
- Products are overdesigned to support future standards
- Cannot iterate on fixes between generations. Must be right the first time.
### Planning Challenges

- Requirements quality varies at the start of the project
- Requirements creep happens
- Initial planning often inaccurate
- Replanning is disruptive
- Causes schedule slips, missed scenarios/use cases

### Increase of Complexity

- Design complexity increases exponentially
- Workforce cannot keep up
- Constrained-random verification doesn’t scale
- Most time spent on debug and coverage closure: These are hard to predict.
- Power and Security are becoming extremely important

### IP Centric Development

- Architecture mindset shift: IP Roadmaps with forward looking requirements
- Reuse and feature superset mentality for design and verification
- Methodology and process update for feature-based, agile development
- Infrastructure update to support this way of working

### Planning for the Unknown

- Increased visibility of development data: Early warning system
- Robust documentation and tracking of requirements
- Using past data to predict the future and plan appropriately
- Building risk into schedules
Hedging Your Bets

Infrastructure expansion and efficiency improvement for better engineering turn-around time: LSF, Compute, Storage

Simulation and Regression time improvement: looking for opportunities to improve performance

Updates to verification strategy and methodology to leverage formal techniques and enable shift left: Formal, HLS

Leveraging EDA state of the art solutions to improve development, debug and coverage closure times

Q & A

...
Abstract
As the verification problem continues to grow, the key metric that many verification teams must closely consider is “Total Verification Throughput.” While verification engines like simulation, formal, emulation and so on have a key part to play in total verification throughput, additional concepts like verification logistics and the utilization of AI can have significant impact and potentially benefit as well. This presentation will introduce the concept of verification logistics and how AI is, and will be, applied.

Biography
Bahadir Erimli is a member of the Cadence Worldwide Field Operations team where, as a Group Director he leads the Verification Applications Engineering team primarily in California including Silicon Valley. Before joining Cadence nearly 12 years ago, Bahadir held a number of senior engineering positions at consume and biotech semiconductor companies. Bahadir is based in San Diego, and holds a Bachelor’s degree in Electrical Engineer from Middle East Technical University in Turkey, as well as advanced degrees in electrical engineer from Caltech.
Verisium AI-Driven Verification

Leverage big data and AI to optimize verification productivity and efficiency
5 Generational Trends... All Anchored Around Compute

The Scale of the Problem is Outpacing Engineering Resources

Total Cost of Silicon – Industry Trend
Amateurs talk about strategy... professionals talk about logistics.

(most commonly attributed to)
Omar Bradley
General, United States Army

Package Throughput = Engines x Logistics

Verification Throughput = Engines x Logistics

Verification Throughput = Engines x Logistics
EDA 1.0

[Diagram showing EDA 1.0]

User Perspective

Meet PPA Goals
Meet Coverage Goals

[Diagram showing user perspective]

Next-Generation EDA

AI-Driven Design and Verification
Multi-engine multi-run learning and optimization

[Diagram showing next-generation EDA]
Extending the Reach of Verification IP

Block level, simulation-based single protocol verification kits

System-level, multi-engine multi-protocol verification apps

- Interface and Memory VIP
- System VIP

- Industry’s broadest portfolio
- Verify compliance and cover the corner cases with TripleCheck
- Highest performance with C-based kernels

- System-level tests up and running in a day
- Validate PCIe® for Arm SBSA
- Boot Linux and Windows over PCIe

- Verify compliance and cover the corner cases with TripleCheck
- Highest performance with C-based kernels

Cadence Verification Solution

- Run, Cover, Debug
- Verisium™ AI-Driven Verification
- Cadence JedAI Data and AI Platform
- Formally, Jasper Emulation
- Xcelium® Simulation
- Xcelium® Prototyping
- Palladium® Virtual Platform
- Helium

Engines
- Compute Arm and x86 CPU
- FPGA Prototyping Protium
- Cadence Silicon Emulation Palladium®

Thank You
Track Session

User Presentations
Lonestar Ballroom – Salon A+B

We would be grateful if you could move to the track session as quickly as possible.
Notes
Abstract
Verification Futures has been running for more than 10 years and in that time more than 25 verification managers have given their views on their main challenges in verification. This talk will summarise those challenges and the main solutions organisations have put in place.
Historic summary

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Mentions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complexity</td>
<td>12</td>
</tr>
<tr>
<td>Debug</td>
<td>12</td>
</tr>
<tr>
<td>Resources</td>
<td>10</td>
</tr>
<tr>
<td>Integrating Methods, Languages and Tools</td>
<td>10</td>
</tr>
<tr>
<td>Completeness</td>
<td>10</td>
</tr>
<tr>
<td>Safety verification</td>
<td>9</td>
</tr>
<tr>
<td>Stability</td>
<td>8</td>
</tr>
<tr>
<td>Mixed Signal</td>
<td>7</td>
</tr>
<tr>
<td>Power Verification</td>
<td>6</td>
</tr>
<tr>
<td>Productivity</td>
<td>5</td>
</tr>
<tr>
<td>Reuse</td>
<td>5</td>
</tr>
<tr>
<td>HW/SW</td>
<td>5</td>
</tr>
<tr>
<td>Security verification</td>
<td>5</td>
</tr>
<tr>
<td>Using AI/ML in verification</td>
<td>4</td>
</tr>
</tbody>
</table>

2022 Wilson Research Group IC/ASIC functional verification trends

- Non-mover: Design complexity stabilising
- Non-mover: 70% projects use UVM
- All other methods <10%
- 44% of IC/ASIC projects are safety-critical
- On the rise: 58% of IC/ASIC projects add security features to their designs
- New entry:...
Notes
Abstract
The highly configurable nature of RISCV ISA makes it uniquely suited for a hierarchical verification methodology covering both architectural and microarchitectural complexity. This technical talk will focus on how Tenstorrent leveraged on the lessons from x86 and ARM to build a modular and scalable CPU verification framework. It will also preview how design complexity has to be tackled looking at silicon as a starting point. And ultimately why robust open source RISCV verification collateral is essential for broader adoption of the ISA from microcontrollers to high performance datacenter class products.

Biography
Divyang Agrawal is a Senior Director at Tenstorrent where he works on RISCV Cores focusing on design verification, emulation, architectural tools and methodologies. He has previously worked on x86 and ARM architectures. Prior to Tenstorrent, Divyang worked at AMD where he held leadership roles within AMD's CPU Cores team working on several generations of high-performance cores. He also led the CPU Power Management IPs and Silicon Validation for all AMD cores. Divyang has a BTech in EE from Nagpur, India and an MBA from University of California at Berkeley.

Slides will be shared during presentation
Abstract
Intel’s 12th generation processors (code named Alderlake) introduced a new asymmetrical design that combines a mix of Performance Cores (P-cores) and Efficient Cores (E-cores), delivering scalable, efficient, multi-threaded performance in a single package. The validation challenge for this asymmetrical design spanned both Pre Silicon and Post Silicon phases. To meet the challenge of validating thoroughly the new asymmetrical design, our validation methodology had to be overhauled; this ranged from updating existing test generators all the way to developing new testing methodologies. In this presentation, we will cover key aspects of our asymmetrical design validation methodology in both Pre and Post Silicon phases, the strategies we adopted and the challenges that we had to overcome.

Biography
Dr. Suneil Mohan received his BE from Anna University in India in 2006 and PhD from Texas A&M University in 2012. He is a senior validation engineer in the Intel E-core team with deep expertise in both Emulation and Post silicon validation. He is currently the Post Silicon debug lead for the E-core team. He has worked on multiple generations of the E-core product line including those that are part of the most recent 13th Generation Intel® Core™ processors. In addition, he has experience working on the ISO26262 standard.
Notes
Agenda

• Introduction to Intel Hybrid architecture
• Pre-Silicon challenges and solutions
• Post Silicon functional validation methodologies
• OS Based Verification
• Functional Validation Sign Off

Intel Performance Hybrid Architecture

Designed to deliver efficient high-compute performance in a large dynamic power and performance range

<table>
<thead>
<tr>
<th>Performance-cores</th>
<th>Efficient-cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>Larger, high-performance cores designed for speed while maintaining efficiency.</td>
<td>Smaller, with multiple E-cores fitting into the physical space of one P-core.</td>
</tr>
<tr>
<td>Tuned for high IPC (instructions per cycle) and high turbo frequencies.</td>
<td>Designed to maximize CPU efficiency, measured as performance-per-watt.</td>
</tr>
<tr>
<td>Supports hyper-threading</td>
<td>Ideal for scalable, multi-threaded performance. Does not support hyper-threading</td>
</tr>
</tbody>
</table>

Pre-Silicon Verification

• 3-prong approach
• Each CPU team performs dedicated IP validation (Simulation, Emulation & FPGA environments)
• SoC validation team performs
  • Integration
  • Hybrid Validation
• Periodic, Consistent check-ins between each Core and the SoC
Extensive Pre-Silicon Microcode Validation

- Built a combined Microcode simulator model.
- Allows validation of interaction before Simulation/Emulation models are built
- Faster turnaround of short experiments for validation

Post Silicon Validation

- Similar approach to Pre Si val
- Each CPU team performs dedicated IP validation
- SoC validation team performs
  - Integration
  - Hybrid Validation
- Periodic, Consistent check-ins between each core family and the SoC

IP Validation teams (E-core and P-core)

- Experts in their uArch
- Validate their IP in isolation
- Support SoC / Hybrid / Integration Debug

SoC Validation team

- Validate Hybrid Integration
- Run Hybrid Workloads
- Verify Key Performance indicators
- Coordinate debug with the appropriate IP team
OS Based Verification

What if Synthetic Workloads are not stressful enough?

- External Software may have corner case behaviors that are not always modelled.
- External Software may do things that are not completely to ‘spec’
- Need to see how random task switching might behave

1. Analyzed Open-Source OS scheduler source code for task switching
2. Ran Task Switching OS Scheduler code on the combined microcode simulator model to understand the behavior
3. Built a randomized OS task switcher for Post Silicon validation

Functional Validation Sign off

Pre-Silicon

- Pass rates for synthetic test content.
- All failures accounted for, understood and dispositioned
- Coverage data analysis complete.
- Power and Perf data collected, analyzed and within expected ranges.

Post Silicon

- Acceptable pass rates for synthetic test content.
- All failures accounted for, understood and dispositioned
- Power and Performance data meeting projections
- 100% pass rate for the OS task switching tests.
- Thread Director working as expected

References and Additional Resources


Lakefield

Meteor Lake

Intel Validation Lab

Intel 12th Gen Validation Platform
We would be grateful if you could move to the track session as quickly as possible.
Notes
Doug Smith
Doulos
Engineer / Instructor

What Can Formal Do For Me?

Gold Sponsor

Abstract
We know formal can prove things, but where do we apply it? Did you know you can use formal to generate simulation testbenches for covering coverage holes or have it visualize your design without writing a single line of testbench code? Formal can be used for identifying metastability, X propagation, fault propagation and detection, equivalence, and so much more. In this tutorial session, we'll have a look at the many ways formal helps out your design verification process.

Biography
Doug Smith is a verification engineer and instructor for Doulos based in the Austin Texas area with expertise in UVM and formal technologies. He has been using formal technology for several decades, performing formal verification on many kinds of designs and formal applications. Likewise, he has provided formal application support at both Jasper and Mentor/Siemens EDA. At Mentor/Siemens EDA, he served as a formal specialist and verification consultant, where he provided both formal consulting and developed an automotive functional safety formal app for performing formal fault campaigns. At Doulos, he delivers training in verification methodologies like UVM, SystemVerilog, and formal technology.

Doug holds a masters degree in Computer Engineering from the University of Cincinnati and a bachelors degree in Physics and Biology from Northern Kentucky University. Currently, he resides in Paige Texas with his wife and family on a small farm where he raises bees, cows, horses, chickens, and pigs and loves driving a tractor.
Global Training Solutions

ESL & Verification Methodology
» SystemVerilog » UVM
» SystemC » TLM-2.0 » Formal

Hardware Design (ASIC / FPGA)
» VHDL » Verilog » SystemVerilog
» Tcl » AMD » Intel FPGA

Arm & Embedded Design
» Arm Cortex A/R/M Series » C » C++
» RTOS » Linux » Yocto » Security

AI & Deep Learning
» Edge AI » Deep Learning
» Python

EDA playground
Practice – Share – Learn
Simulate your hardware description code in a web browser for free

Call +1-888-GO-DOULOS to discuss your training needs
www.doulos.com
What Can Formal Do For Me?

- What is formal?
- Where can formal be used?
- Applications for formal
- Wrap-up

What is Formal?

*Formal verification uses mathematical formal methods to prove or disprove the correctness of a system’s design with respect to formal specifications expressed as properties...*  

Formal ...
- Is mathematical and algorithmic
- Proves the correctness of a design
- Guarantees the implementation meets requirements
- Requires no testbench or stimulus

Simulation vs Formal

Simulation
Tests the design
Testbench generates all stimulus and performs checking

Test input stimulus \[\xrightarrow{\text{DUT}}\] Testbench checker

Formal
Proves the design meets the requirements
Requirements become formal target
Formal generates all input

Formal input \[\xrightarrow{\text{DUT}}\] Assertion targets
What Can Formal Do For Me?

- What is formal?
- Where can formal be used?
- Applications for formal
- Wrap-up

Formal Throughout the Design Cycle

- Architecture Planning
- Design
- Verification
- Sign-off
- Post-silicon

- Architectural modeling
- Processor ISA compliance
- Verify spec
- Automatic design checking
- Design exploration
- CDC / RDC
- Model checking
- Interface VIP
- Reachability
- Equivalence
- Coverage
- Assertion quality
- Test plan generation
- Post-silicon debug
- Verifying ECOs

What Can Formal Do For Me?

- What is formal?
- Where can formal be used?
- Applications for formal
- Wrap-up

Applications for Formal

- Design exploration
- Automatic design checking
- Model checking
- Reachability
- Equivalence
- Sign-off
- Post-silicon
**Design Exploration**

unique case (State)
- Zero: if (Buttons[1]) NextState = Start;
  - Start: begin
    - WatchRunning = 1;
  - Running: begin
    - WatchRunning = 1;
      - if (Buttons) NextState = Stop;
  - Stop: if (!Buttons) NextState = Stopped;
  - Stopped: if (Buttons[1]) NextState = Start;
    - else if (Buttons[2]) NextState = Reset;
  - Reset: begin
    - WatchReset = 1;
      - if (!Buttons) NextState = Zero;
    - end
  - endcase

cover property ( State == Stopped );

**Formal Generated Trace**

cover property ( State == Stopped );

Design visualization with ...
- No testbench
- No testcase

**Auto Trace from Coverage App**

Select line to reach
- Generate example trace
- Generate wave

**Draw a Scenario**

Draw trace
- Generate wave
Applications for Formal

- Design exploration
- Automatic design checking
- Model checking
- Reachability
- Equivalence
- Sign-off
- Post-silicon

Automatic Property Checking

- Array bounds
- Arithmetic overflow
- Priority and unique case
- Set and reset both active
- Reachable X assignment
- Deadlock / livelock
- Incomplete sensitivity lists
- ... and others

Array Bounds Check

```verilog
logic [7:0] address;
logic [0:3] array;
int k, n;
assign n = address >> 6;
always @(
  posedge clock)
if (write)
  array[address] <= data_in;
else if (read)
data_out <= array[n];
else
  data_out <= array[k];
c_k: assume property ( @(
    posedge clock) k >= 0 && k < 4 );
```

Arithmetic Overflow Check

```verilog
logic [7:0] address;
always @(
  posedge reset or posedge clock)
begin
  logic [3:0] sum;
  if (reset)
    sum <= 0;
  else
    sum <= sum + address;
end
assert property ( @(
    posedge clock) disable iff (reset)
    sum + address < 16 );
```
### Unique Case Check

```verilog
logic sel, c1, c2,
always @(posedge clock)
unique case (sel)
0: out2 <= 0;
1: out2 <= 1;
endcase
always @(posedge clock)
unique case (sel)
c1: out3 <= 0;
c2: out3 <= 1;
endcase

assert property (@(posedge clock) c1 | c2);
assert property (@(posedge clock) !(c1 & c2));
```

宦, 并且 并行 案 一时 一时

### Other Automated Checking

- Clock-domain crossing (CDC)
- Reset-domain crossing (RDC)
- Low-power UPF checks
- Glitch checking
- ... and others

### Applications for Formal

- Design exploration
- Automatic design checking
- Model checking
- Reachability
- Equivalence
- Sign-off
- Post-silicon

### Model Checking

Formal uses SVA for checking requirements

```verilog
assert property (!WE & OE);
assert property (Size <= Max);

property incr_size;
    int sz;
    (Wr, sz = Size) @1 !Ready[*1:$] @1 Ready -> Size == sz + 1;
endproperty
assert property (incr_size);
```
Capturing a Specification

After a start pulse, stop must go true on the next or second clock, and must remain true for exactly two clocks.

Example traces:

assert property (start |-> ##[1:2] stop[*2]);

too short
too long
too late

Prove Protocol Correctness

Bit [2:0] start_k, stop_k;
always @ (posedge clk) begin
if (start)
start_k <= start_k + 1;
if (stop)
stop_k <= stop_k + 1;
end

property overlap_start_stop;
bit [2:0] k;
(start, k = start_k) |
- > ##[1:4] stop && (stop_k == k);
endproperty

assert property overlap_start_stop;

new variable for each instance of property
local variable assignment

End-to-End Checking

Design exploration
Automatic design checking
Model checking
Reachability
Equivalence
Sign-off
Post-silicon

Applications for Formal
What Is Reachability?

Reachability – given any legal stimulus, is it possible to reach a scenario or line of code?

cover property ( State == Stopped );

Many Applications

Deadlock  X-propagation
Livelock   Connectivity
Vacuous assertions Registers
Liveness   Security

Liveness

Does something eventually happen?

assert property ( a |-> s_eventually ( b ) );

clk
a
b

Hard (impossible) to prove in simulation

X Propagation

Non-resettable flops

cover property ( $isunknown( dataout ) );

Non-resettable flops
assert property ( processor.PCLK == interconnect.PCLK );

Limited key access?
Keys unreachable from other paths?

Provable by formal

assert property ( sel & addr == 0x0 |-> soc.dma.ctrl == ... );

Applications for Formal

Design exploration
Automatic design checking
Model checking
Reachability
Equivalence
Sign-off
Post-silicon
Logic Equivalency Checking

RTL versus gate-level netlist
Netlist versus netlist
Only works with recognizable equivalency points (signal names)

```
module selAB (input logic clk, input logic QA, selA, QB, selB, output logic Q);
always @(posedge clk)
begin
  if (selA) Q <= QA;
  if (selB) Q <= QB;
end
endmodule
```

Sequential Equivalency Checking

Dynamic, not static like LEC – advances the clock
Shows equivalency between different implementations
Equivalency at the port-level
RTL <-> RTL, RTL <-> HLS (SystemC/C/C++)

Many Applications

VHDL <-> Verilog translation
Incremental feature updates (chicken bits)
ECO fixes
Data path verification
C to RTL equivalence
Functional safety
Fault injection
Safety mechanism insertion

Fault Injection

SEC can traverse through state better than model checking
Simply check if outputs are affected by the injected fault
**Functional Safety**

```verilog
// Find residual fault(s)
cover property ((fault == 1) && violation && !detected);
```

**Data Path Verification**

```verilog
// RTL module fmul (...) { input logic [SIZE-1:0] multiplier,
// input logic [SIZE-1:0] multiplicand,
// output logic [SIZE-1:0] product,
// ...
```

**Applications for Formal**

- Design exploration
- Automatic design checking
- Model checking
- Reachability
- Equivalence
- Sign-off
- Post-silicon

**Areas Formal Helps Sign-off**

- Achieving coverage closure in simulation
- Creating simulation testbenches to hit coverage holes
- Measure assertion quality
- Formal coverage
- Testplan generation
- Reachability
Coverage Exclusions for Simulation

Formal finds unreachables and generates exclusions:

```bash
<formal_tool> generate exclude exclude_file.tcl

coverage exclude -scope
[/tb_axi4lite_2_apb4/dut/u_master_interface/u_apb_master_s
 c] -srcfile .../src/vlog/apb_master_sc.v -linerange 88 -item x 1 -reason "EU"

coverage exclude -scope
[/tb_axi4lite_2_apb4/dut/u_master_interface/u_apb_master_s
 c] -srcfile .../src/vlog/apb_master_sc.v -linerange 106 -item x 1 -reason "EU"
```

Testbench Generation

Generate stimulus to target coverage holes:

```bash
<formal_tool> generate testbenches

module replay_vlog;
initial begin
$1;
force axi4lite_to_apb4.use_1clk_i = 1'b0;
force axi4lite_to_apb4.PRESETn_i = 1'b0;
force axi4lite_to_apb4.PREADY_i = 1'b0;
force axi4lite_to_apb4.PSLVERR_i = 1'b0;
force axi4lite_to_apb4.PSELx_i_csr = 1'b0;
...
```

Measuring Assertion Quality

RTL Mutation Coverage

```
Non-detected
(no assertions fail)
```

Formal Coverage

Assertion density – are there enough?
Cone-of-influence (COI) coverage
Proof core coverage
Code coverage
Proof core coverage
Functional coverage
Cover properties
Synthesizable covergroups
Assertion quality
Mutation coverage

Merges with simulation coverage
Applications for Formal

Design exploration
Automatic design checking
Model checking
Reachability
Equivalence
Sign-off
Post-silicon

Post-Silicon Debug

Formal can reproduce post-silicon results for debug

Constrain formal to pin values

assume property ( pins == ... );
cover property ( state == ERROR );

What Can Formal Do For Me?

• What is formal?
• Where can formal be used?
• Applications for formal

What Can Formal Do For Me?

• What is formal?
• Where can formal be used?
• Applications for formal

Summary

Formal complements your simulation flow
Formal verifies scenarios hard or tedious in simulation
Formal can be part of any verification planning and effort
Why would you not take advantage of what formal can do?
We hope you found this information helpful!

Thank you for attending.
We would be grateful if you could move to the track session as quickly as possible.
Notes
Peter Grove & Steven Holloway

Renesas
Distinguished Member of Technical Staff &
Member of Technical Staff

Renesas’s Submission to the UVM-(A)MS working group

User Paper

Abstract
Explanation on how UVM can be applied to DMS/AMS using a concept of a MS Bridge module. The focus will be on an AMS Device-Under-Test, but the concepts work for DMS. The audience will be guided over subtleties of AMS simulators and a known limitation with the proposal and possible solutions. There will be a walk through of how this was applied to a Mixed signal block. The audience should not take away the current implementation until an official release of UVM-AMS has been made. The current contents of the presentation and example code has been shared to the EDA community to feed into a white paper on the topic. Steven will cover the UVM aspects and Peter will go over mixed signal parts.

DMS – Digital-Mixed-Signal also referred to as Real-Number-Modelling
AMS – Analog-Mixed-Signal
MS – Mixed Signal

Biography
Peter Grove
Peter has worked in the industry starting back in 2001 when he joined a small company called Wolfson MicroElectronics, where he was project lead for more than 15 production devices. Since then Peter has only worked at one other company, Nujira, before joining Dialog (now Renesas) at their Edinburgh office. Peter has been with Dialog since 2014. Peter’s background has been main digital design, but has over the years taken charge of many large mixed signal devices that are in volume production and been exposed to enough analogue design work to appreciate the issues they face in verification. Peter has an eye for looking for ways in which techniques can be done to improve chip level coverage, simulation runtime improvement to name a few.
Peter is also in a unique position that during his days at Wolfson he was a key player in defining their schematic/Layout tool set with integrated revision control. This has allowed Peter to gather a large number of skills not just in design work but in all the backend flows and EDA tools, understanding different netlist types and how the tools work.

Peter’s technical interests are mixed signal and analogue verification methodologies, design flows. Peter also is an Acellera SystemVerilog-AMS committee chair, UVM-AMS member/key contributor making sure the ‘users’ feedback on the language is considered and not what the vendors just want to support.

**Steven Holloway**

Steve has 20+ years’ experience of digital verification including eRM, OVM, UVM and formal property checking. He has led the verification of large-scale consumer SoC projects. He joined Dialog Semiconductor in 2011 and previously worked for Doulos, NXP and Trident Microsystems. He joined the Technical Ladder in 2015. Steve has presented at multiple external conferences including a panel session at DVCon US. He participates in industry standards bodies and has contributed code to the Acellera UVM-AMS working group.
**Agenda**

- Why UVM-MS
- Verilog-AMS Simulator DC OP / Transient behavior
- UVM MS Bridge to analog resource (UVM->AMS/DMS Connection)
- UVM-MS Phasing Requirement
- UVM messaging from AMS files and $root cells

**WHY UVM-MS**

- UVM is the industry standard methodology for reusable metric driven verification
  - UVM-MS is the standardisation of analogue/mixed signal extensions for UVM
    - Allow UVM to be more mixed-signal aware
    - Improved verification of analogue/mixed signal designs
    - Same degree of thoroughness for both analogue and digital parts
- Originally named UVM-AMS but focus is to support any MS system; DMS, RNM, Spice or a mixture
  - Metric-driven verification suits following objectives due to verification space size
    - Verifying analogue performance under large set of digital configurations
    - Digital control system transitions interacting with analogue functions
    - Dynamic control between analogue & digital circuits under wide range of conditions
    - Finding problems with A/D interaction in unexpected corner cases
- Randomisation is not mandatory and benefits are gained even when using directed tests
  - Standard methodology
  - Plug & play reuse of existing UVM components
  - Rich debug & messaging scheme integrated with simulator

**UVM-MS REQUIREMENTS**

- Apply UVM methods and techniques to AMS circuits and systems while allowing CMS/RNM.
- Enable a single environment to work whether it is CMS/RNM or AMS by changing the abstraction of the DUT.
- Extend the use of UVM components, and extensions thereof, into the physical layer enabling AMS verification.
- Allow predictable coordination of stimulating/measuring a signal
- Adhere to the sequence/sequence-item mechanism used by UVM
  - Independent of the abstraction level of the AMS signals (electrical, PINM, UDT, etc.)
- Eliminate the need to rely on conversion elements to change the abstraction level of the DUT signals.
- Use existing language standards; SV and Verilog-AMS
  - Changes too years to get agreement.
Why UVM

Understanding of UVM

E.g. Make a Cap open for a particular test before DC op or short/open a path saving multiple testbenches of configurations.

Digital to analogue events cause matrix re-evaluation and timestep backtrack.

Skipping DC op will cause odd results and vendor specific.

UVM messaging from AMS files and $root cells

UVM MS Bridge to analog resource (UVM->AMS/DMS Connection)

UVM MS Phasing Requirement

UVM messaging from AMS files and $root cells

To avoid race conditions between digital blocks.

To make sure the correct value is captured between the transient and steady state operating point of all the nodes/branch currents.

To avoid odd issues at initialization of the simulation finishes.

MS Phasing Requirement

AMS Simulator DC OP / Transient behavior

VERILOG-AMS SIMULATOR SCHEDULING – TIME 0

Variable initialization

All initial blocks executed till they consume time.

Order of initial blocks non-deterministic.

UVM phases (e.g. run_phase) included.

DC Operating Point at time Zero

VERILOG-AMS SIMULATOR SCHEDULING - TRANSIENT

Some digital to analogue event.

Digital to analogue events cause matrix re-evaluation and timestep backtrack.

Most simulators save any digital var in the analog block as a D2A to monitor.

Agenda

▪ Why UVM-MS
▪ Verilog-AMS Simulator DC OP / Transient behavior
▪ UVM MS Bridge to analog resource (UVM->AMS/DMS Connection)
▪ UVM-MS Phasing Requirement
▪ UVM messaging from AMS files and $root cells

Digital initial block(s)

Analog initial block(s)
### Variables are ‘owned’ by one engine, but can be read by another.

- AMS cannot access digital variables that are dynamic. (Everything in the matrix is fixed at time 0)
- Generally avoid ‘string’ datatypes in Verilog
- 'reals' for monitored continuous signals.
- Enable logic strength/ports on wires.

### Agenda

- Why UVM-MS
- Verilog-AMS Simulator DC OP / Transient behavior
- UVM MS Bridge to analog resource (UVM->AMS/DMS Connection)
- UVM-MS Phasing Requirement
- UVM messaging from AMS files and $root cells

### MIXED SIGNAL BRIDGE & ANALOG RESOURCE

- MS Bridge (SystemVerilog) to connect the UVM layer to the analog resource.
  - The analog resource could be DMS/RNM/AMS based on DUT pin abstraction.
- Proxy Features (mandatory)
  - DMsh: Connect user needed for logic strength by some digital type IO's only logic are valid.
  - Push analog-resource controls using function calls.
  - Pull analog-resource limits by end of transition detection or other synchronization from resource control.
  - E2C analog-resource values via functions calls.
  - OOMR: Work-around for monitored continuous signals.
- SystemVerilog Interface (optional) digital signals as they are currently
  - Enable logic strength points on wires.
- More suited to allow reuse of existing IF with a MS Bridge.
- Plug-in/MSRES, VDDREF, VSS set as 'interconnect' in MS Bridge.
- Allow shared DMS/AMS/RNM analog resource.
- REF_VDD/REF_VSS for SV IF logic-level to electrical conversion.
- OOMR's work around datatype limitations on AMS modules I/O's.

### ANALOG RESOURCE – WHAT DOES IT LOOK LIKE

Ensure OOMR, port, parameters from MS Bridge to analog_resource abstractions are the same!
For logic signal the (.PLUS( Agent could use override using the UVM factory to extend the pure digital solution to a mixed signal one.

Classical RNM would drive real numbers from UVM sequence/driver within the agent.

In AMS this would generate many D2A events or not give enough finesse to the signal.

Place the signal generator is located in the domain of the DUT I/O it will connect to.

A sine wave is made up of 4 properties: frequency, phase, amplitude, and DC bias.

UVM transaction encodes the properties of the sine wave as real values in the uvm_sequence_item

Properties passed to analog_resource to generate the sine waves.

Still honors the UVM paradigm of having a relatively simple interface for the test writer.

Change from classical UVM sequence/drivers for UVM-AMS.

Proxy Template (API)

Proxy class derived from i_cap_bridge

Proxy instance in bridge module extends resource implementation

Implementation of proxy API methods in bridge module in turn execute analog resource “core”
methods – hence “proxy” pattern.

Proxy MUST always be present and instanced as __uvm_ms_proxy – more later on this!

SV IF is optional but must be placed in ms_bridge.

Agent could use override using the UVM factory to extend the pure digital solution to a mixed signal one.
Analog generation update

• Transition filter in AMS is used to convert discrete signals to a continuous time scale.

Target is different from

If target is different from

Start/Stop have tolerances!

DUT is logic

DUT is AMS

Examples to be provided!

• Dynamic or static supplies.
• Control all aspects.
• Examples to be provided!
Agenda

- Why UVM-MS
- Verilog-AMS Simulator DC OP / Transient behavior
- UVM MS Bridge to analog resource (UVM->AMS/DMS Connection)
- UVM-MS Phasing Requirement
- UVM messaging from AMS files and $root cells

UVM PHASING REQUIREMENTS FOR AMS

- MS Bridges will have parameters.
- UVM should have a means to read/modify/write params before simulation consuming time
- Implement methods getParameter() / setParameters() in proxy
- Use existing UVM phases to guarantee read/modify/write order

<table>
<thead>
<tr>
<th>UVM Phase</th>
<th>What should happen for MS resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>start_of_simulation_phase</td>
<td>Read parameters values from 'SV+VAMS' module (Instrument/Passive) into the agent's configuration</td>
</tr>
<tr>
<td>connect_phase</td>
<td>Apply agents parameters based on test requirements</td>
</tr>
<tr>
<td>run_phase</td>
<td>Start execution, 'SV+VAMS' module (Instrument/Passive) begins execution</td>
</tr>
<tr>
<td>end_of_simulation_phase</td>
<td>Set VAMS initial parameter values and to have if($realtime &lt;= 0.0) #1step; to cause a DC OP to happen.</td>
</tr>
</tbody>
</table>

ANALOG RESOURCE CONFIGURATION

- Analog components tend to be placed with initial values as parameters. E.g. A decoupling cap on a LDO output.
- Test cases can override the configuration, which are then set in the analog resource in start_of_simulation_phase.
- This is pre DC OP as you can do step changes to analog values!

AMS START-UP & UVM PHASING

- AMS models will have parameters!
- UVM should have a means to read/modify/write params before simulation consuming time
- Use UVM phasing to guarantee read/modify/write order

Variable used in rest of code not parameters

Parameter assigned to variable so setParameters can override them.
AMS STARTUP IN UVM RUN_PHASE()

Raise test objection →Wait for DC-OP →Launch sequences

Ensures time is consumed

virtual task my_ams_test.run_phase(uvm_phase phase);
  phase.raise_objection(this); // Prevent termination in DC-OP
  // ... (remaining code is not shown)
  phase.drop_objection(this); // Test termination
endtask:

Agenda

▪ Why UVM-MS
▪ Verilog-AMS Simulator DC OP / Transient behavior
▪ UVM MS Bridge to analog resource (UVM->AMS/DMS Connection)
▪ UVM-MS Phasing Requirement
▪ UVM messaging from AMS files and $root cells

UVM MESSAGING REQUIREMENT

▪ Need to filter and control generation of messages from analog resource
▪ UVM offers this control for components in UVM hierarchy
▪ But analog resource is not part of the UVM component hierarchy. It’s a module!
▪ However, if we extend the MS proxy from uvm_report_object
  ▪ set_report_handler() can redirect handling to the enclosing UVM MS monitor
  ▪ messages from MS bridge (and below) can use the proxy context
  ▪ uvm_info_context(...., report_handler) takes reporting object to provide context
  ▪ Messaging macros called from analog resource can use upscoping (see later)
  ▪ Recommend to include %m in the UVM message body to get a physical path.

MS MESSAGING CONTEXT

virtual function void uvm_ms_monitor::connect_phase(uvm_phase phase);
  // ... (remaining code is not shown)
endfunction:

UVM_INFO @ 0.000 ns: uvm_test_top.env.ms_agent.monitor [MS_MONITOR] Incorrect bias voltage: 0.125V
UVM REPORTING FROM ANALOG RESOURCE.

- UVM-reporting macro not supported in Verilog-AMS modules
- Use analog domain to detect the issue and toggle an integer.
- UVM_INFO ../../includes/SV Bridge

UVM MESSAGE – VERILOG-AMS ANALOG BLOCK

- Use analog domain to detect the issue and toggle an integer.
- Use _context reporting macros to direct message to relevant component.
- Use `context reporting macros to direct message to relevant component

UVM MESSAGE

Analog Resource

Hence the proxy name requirement!

UVM AMS REPORTING ISSUES

- Use analog domain to detect the issue and toggle an integer.
- Use _context reporting macros to direct message to relevant component
- Use _context reporting macros to direct message to relevant component

Table:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>volts</td>
<td>real</td>
<td>0.12300000</td>
</tr>
<tr>
<td>amps</td>
<td>real</td>
<td>0.56700000</td>
</tr>
<tr>
<td>henrys</td>
<td>real</td>
<td>0.00000000</td>
</tr>
<tr>
<td>farads</td>
<td>real</td>
<td>1e-09</td>
</tr>
<tr>
<td>radians</td>
<td>real</td>
<td>100</td>
</tr>
<tr>
<td>degrees</td>
<td>real</td>
<td>180</td>
</tr>
<tr>
<td>radians</td>
<td>real</td>
<td>90</td>
</tr>
<tr>
<td>degrees</td>
<td>real</td>
<td>0</td>
</tr>
<tr>
<td>radians</td>
<td>real</td>
<td>45</td>
</tr>
<tr>
<td>degrees</td>
<td>real</td>
<td>22.5</td>
</tr>
</tbody>
</table>

- Use _context reporting macros to direct message to relevant component
- Use _context reporting macros to direct message to relevant component
Two $root cells

• test_case to encapsulate the stimulus generation SV Class based world.
• test_env is independent of how the test_case is setup enabling a DMS/AMS class based environment.
• test_env could be a schematic and analog resources driven or test View
• Proposed system allows parameters to be used.
• Proposed system worth hardware accelerators for the physical design.

RELATED RMS SETUP

• Logic Conversion needs reference VDD/VSS levels.
  1. Dynamic tracking
     1. Dedicated pins REF_VDD/REF_VSS in MS_BRIDGE/Analog Resource.
     2. OOMR using analog_node_alias pin parameters for AMS only
     + Can only be parameters as this is setup pre DC OP.
     • Ideally it would use ref_vdd/vss but alias to port is not allowed. (Verilog AMS LRM 9.20)
  2. real values set like other controls in the MS Bridge to the analog resource.

PROVIDED PACKAGES/INCLUDE FILES

<table>
<thead>
<tr>
<th>Statement</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>import uvm_ms_pkg;</td>
<td>within the MS Bridge and uvm_ms_agent.</td>
</tr>
<tr>
<td>`include &quot;uvm_ms.svh&quot;</td>
<td>For Verilog AMS modules defined on the analog_resource or hierarchy.</td>
</tr>
<tr>
<td>`include &quot;uvm_ms_proxy.svh&quot;</td>
<td>For AMS modules defined on the analog_resource or hierarchy. For example AMS vams or SV module this include would allow the AMS module to use the same messaging system.</td>
</tr>
<tr>
<td>`include &quot;uvm_ms.svh&quot;</td>
<td>For inclusion in the MS Bridge to enable the communication from the analog_resources. It requires the AMS Proxy instance is named __uvm_ms_proxy.</td>
</tr>
</tbody>
</table>

LOGIC CONVERSION WITHIN ANALOG RESOURCE

© 2023 Renesas Electronics Corporation. All rights reserved.
At any one time there is one driver plus the pull1 component.
Logic strength rules control the resolved value.

Classical UVM where the pad is a digital model.
SV IF can have ports and those ports are wires which enabled Verilog Strengths.
Allow input for I2C SDA.
For UVM-MS the ideal solution is to use the SV IF without modification.
Could be multiple slaves.

Electrical driver creates a feedback loop onto the logic net.
Detection of Z-state on the electrical net not possible.
Different feedback drivers strength allowed but which should be used?
Verilog-MS DRS not available in modules?
CM could be used but then it depends on what else the driver needs to do.
Logic strength not natively accessible.

Split Interface inout pins into input/output pairs, thus replicating DRS system.
Not ideal but removes the feedback loop.
Use VPI routine to get logic value/strength of logic net been driven to 'electrical driver'.
8bit bus, 2 bits for value, 3 for logic 1's strength and 3 for logic 0's strength. Similar to $drive_strength from Verilog-AMS.
Strength changes output resistance.

Independent organization founded in 2000.
Mission to collaborate to innovate and deliver global standards that improve design and verification productivity.
Partnership with IEEE for formal standardisation & governance.
Renesas has representatives on many working groups, including UVM, UVM-MS, SV-AMS, SystemC.
Larry Lapides

Imperas Software Ltd
VP Worldwide Sales

A Modern Fable: The Lost Art of Processor Verification

Platinum Sponsor

Abstract
The open standard Instruction Set Architecture (ISA) of RISC-V offers new design flexibilities and opportunities, and is having a significant impact on the design side of many SoC projects. An optimized processor enables developers to unlock hidden value in performance, power savings, security, differentiated features, and an enduring market advantage.

While every SoC design team now has a free architecture license to build a custom RISC-V processor or extend an existing core with custom instructions, this also represents a surge in verification work and a step-change in verification complexity. With other ISAs, verification methodologies have largely been kept proprietary. Now within the RISC-V community, the art and science of processor verification is resurfacing. This represents a massive migration in verification responsibility, and the creation of a new verification ecosystem.

This talk outlines the various methodologies for RISC-V processor verification, which leverage established SoC verification technologies with UVM and SystemVerilog. The individual components of a step-compare methodology will be discussed, including reference model, verification IP, functional coverage and test generation. Detailed examples of successful, complex processor verification projects will be presented, including flows to support verification of complex events and architectures such as interrupts, Debug and privilege modes, multi-hart processors and multi-issue and out-of-order pipelines.
Biography
Larry is currently VP Worldwide Sales at Imperas Software Ltd., and previously ran worldwide sales at EDA companies including Verisity Design (the top performing IPO of 2001 in the U.S.). Larry has about 30 years in software tools and EDA, plus time spent in infrared sensors and systems engineering. Larry holds a BA in Physics from the University of California Berkeley, a MS in Applied and Engineering Physics from Cornell University and a MBA from Clark University where he was an Entrepreneur-in-Residence during Fall 2006, when he developed and taught the course on Entrepreneurial Communication and Influence.
Notes
ImperasDV™ RISC-V Processor Verification Solutions

RISC-V is an open standard ISA (Instruction Set Architecture) that allows any developer to design and extend a custom processor, while remaining compatible with the growing ecosystem of supporting tools and software. The innovation and impact of RISC-V on the design side is driving new developments across all market segments and applications.

Now, with ImperasDV, developers have a dependable, reference model-based solution for verification that is compatible with the current UVM SystemVerilog methods for SoC verification.

www.imperas.com/imperasDV

ImperasDV and Imperas RISC-V processor verification technology is already in active use with many leading customers, some of which have working silicon prototypes and are now working on 2nd generation designs. These customers, partners, and users span the breadth of RISC-V adopters from open source to commercial, research to industrial, microcontrollers to high-performance computing.

A select sample of these include Codasip, Dolphin Design, EM Microelectronics (Swatch), Frontgrade Gaisler, Intrinsix, NSITEXE (Denso), Nvidia Networking (Mellanox), NXP, OpenHW Group, MIPS, Seagate Technology, Silicon Labs, Valtrix Systems, and Ventana Micro Systems, plus many others yet to be made public.

RISC-V Processor Functional Verification with RVVI & ImperasDV

SystemVerilog top level Testbench

- UVM env (optional)
- Simulation control
- RISC-V Core RTL (DUT)
- mem
- test

ImperasDV

- riscvSACOV functional coverage
- trace2cov
- trace2api
- trace2log

RVVI-TRACE

- C/C++

System Verilog

- Configuration
- State comparison
- Pipeline synchronization
- Scoreboard
- Pass/Fail Determination

RISC-V Processor Design Verification

- Verification IP
- Test & Coverage suites
- RISC-V Reference model
Agenda

• RISC-V and processor verification
• RISC-V processor models
• RISC-V processor verification methodology
• Processor verification success
• Summary

RISC-V Is Why We Are All Worried About Processor Verification

• RISC-V is taking over the processor world, except for x86
  • Yes, that includes Arm
• RISC-V processor customization means that every RISC-V developer needs to verify the RISC-V processor
• Lost art? Processor IP vendors guard their verification methodology and details more than the IP itself
  • With the verification flow, someone could reverse engineer a high-quality processor
  • There are few public details about x86, Arm or Apple processor verification
RISC-V Freedom Enables Domain Specific Processing

- **Who**: RISC-V users include traditional semiconductor companies, and embedded systems companies now practicing vertical integration by developing their own SoCs.
- **What**: RISC-V is an open instruction set architecture (ISA), it is not a processor implementation.
- **Where**: RISC-V is growing in market segments where x86 (PCs, data centers) and Arm (mobile) architectures are not dominant.
  - Small microcontrollers for SoC management, replacing proprietary cores
  - Verticals such as IoT and automotive
  - Horizonal markets such as security and AI/ML
  - Deep-embedded applications
- **Why**: The freedom of the open ISA enables users to develop differentiated domain specific processors and processing systems.

Keys to RISC-V SoC Success

1) Processor IP
   - Processor IP vendor
   - Open source IP
   - Build it yourself
2) Processor verification
3) Software porting, development, bring up, test

- All 3 areas need to account for the addition of custom features to the processor (because everyone adds custom features to the processor)

RISC-V Processor Complexity

- RISC-V is a modular instruction set architecture.
- Any extension (functional group of instructions, e.g. atomics, compressed, floating point, vector) can be added to the base processor.
- Then add in interrupts, privilege modes, Debug mode, multi-hart (multi-core), etc. and it gets complex.
- Then processor DV, tool chain development and other software development is needed.
RISC-V Processing Subsystems

- Multi-processor subsystems are commonly being developed using RISC-V cores
- Application areas include DSP, AI/ML and packet processing
- This adds complexity to both the DV and software development tasks

Agenda

- RISC-V and processor verification
- RISC-V processor models
- RISC-V processor verification methodology
- Processor verification success
- Summary

RISC-V Model Requirements

- Model the ISA, including all versions of the ratified spec, and stable unratified extensions
- Model other behavioral components, e.g. interrupt controllers
- Easily update and configure the model(s) for the next project
- User-extensible for custom instructions, registers, ...
- Model actual processor IP, e.g. Andes, MIPS, NSITEXE, OpenHW, SwRV, ...
- Well-defined test process – for the model – including coverage metrics
- Interface to other simulators, e.g. SystemVerilog (Xcelium), SystemC (Helium), Imperas virtual platform simulators
- Interface to software debug tools, e.g. GDB/Eclipse, Imperas MPD
- Interface to software analysis tools including access to processor internal state, etc.
- Most RISC-V ISSs can meet one or two of these requirements
- Imperas models and simulators were built to satisfy these requirements, and matured through usage on non-RISC-V ISAs over the last 15+ years

Imperas OVP RISC-V Fast Processor Models

- Use cases
  - Architecture analysis, including (especially) custom instructions
  - Software development, debugging and test
  - Processor and SoC verification
- Existing Imperas Open Virtual Platforms (OVP) Fast Processor Models of ...
  - Generic or envelope models of RISC-V RISC-V IMAFDCEVBHKPZ* M/S/U privilege modes
  - Models of processor IP vendors: Andes, MIPS, NSITEXE, OpenHW, SwRV, ...
  - Models for developer building their own processor
- Custom instructions easily added by user or by Imperas
- New instructions are added in a side file so as not to perturb the verified model
- Custom instructions are analyzed for effectiveness
- Models are built using Test Driven Development (TDD) methodology
  - Tests are built at the same time as features are added
  - Continuous Integration (CI) test flow used
  - $50,000 tests for models + simulator
- Additional testing by processor IP vendors to validate models

Dolphin Design “Panther” DSP

Imperas OVP Data Flow Processor

© 2023 Imperas Software Ltd.
Models Drive Customization

- Custom instructions are added to optimize a specific application or set of applications within a domain
- Models let you explore quickly
  - Much faster to develop than RTL
  - Better profiling information available
  - Easier to debug software
- Methodology
  - Start by characterizing the application to be optimized
  - Then add the custom instructions, evaluate, and iterate

Agenda

- RISC-V and processor verification
- RISC-V processor models
- RISC-V processor verification methodology
- Processor verification success
- Summary

5 Levels of RISC-V Processor DV Methodology

1) Hello World
2) Self-checking tests (e.g. Berkeley torture tests pre-2018)
3) Post-simulation trace log file compare
4) Synchronous step-and-compare
5) Asynchronous continuous compare
**5 Levels of RISC-V Processor DV Methodology**

1) Hello World
2) Self-checking tests (e.g. Berkeley torture tests pre-2018)
3) Post-simulation trace log file compare
4) Synchronous step-and-compare
5) Asynchronous continuous compare

**3) Post-Simulation Trace Log File Compare (Entry Level DV)**

- Process
  - use random generator (ISG) to create tests
  - during simulation of ISS write trace log file
  - during simulation of RTL write trace log file
  - at the end of both runs, run logs through compare program to see differences / failures

- ISS: `riscvOVPsimPlus` includes Trace and GDB interface

- ISG: `riscv` from Google Cloud / Chips Alliance

**5) Async Continuous Compare (Highest Quality DV Methodology)**

- Asynchronous events are driven into the DUT
- Tracer informs reference model about async events
- Verification IP handles async events, scoreboard, comparison, pass/fail
- Asynchronous continuous compare methodology is needed to support features such as interrupts, privilege modes, Debug mode, multi-hart, multi-issue and DoC pipeline, ...

**ImperasDV Components**

- Reference model needed for comparison of correct behavior
- Verification IP provides ease of use, saves time and resources
- RVV standard provides communication between test bench and Reference
- `riscvSADDV`: Functional coverage modules
- Test suites: `riscvSADDEV`, directed test suites for difficult extensions
- Multi-Processor Debugger (MPD) enables RTL-reference model comparison

- Feature selection and design choices require serious consideration due to implications of every decision
  
- Every addition dramatically compounds verification complexity
  
- Before 2023, no off-the-shelf tools/products available for DV of processors ... then came ImperasDV
  
- ImperasDV, with async continuous compare methodology, is needed to support features such as interrupts, privilege modes, Debug mode, multi-hart, multi-issue and DoC pipeline, ...
**ImperasDV: Verification IP**

- Data prep for functional coverage
- Data movement from SystemVerilog to C verification IP
- Logging data

**Open Standard RISC-V Verification Interface: RVVI**

- RVVI = RISC-V Verification Interface
  - [GitHub](https://github.com/riscv)
- Work has evolved over 3 years
  - Imperas, EM Micro, SiLabs, OpenHW
- Standardize communication between testbench and RISC-V VIP
- Two parts (currently):
  - RVVI-TRACE: signal level interface to RISC-V VIP
  - RVVI-API: function level interface to RISC-V VIP

**RVVI-TRACE Enables DUT Introspection by Verification IP**

- Defines information to be extracted by tracer
  - SystemVerilog interface
    - e.g. interrupts, debug requests
- [GitHub](https://github.com/riscv-verification/RVVI/tree/main/RVVI-TRACE)

**RVVI-API Enables Easy Implementation of Step-Compare Flow**

- Reference model encapsulation
  - Includes DUT reference state storage
  - Includes synchronization technology
    - Can sync, async, interrupts, debug, multi-core
  - Pipeline synchronization is key for asynchronous event DUT
  - Includes comparison technology
    - Comparisons are done on DUT/Reference Model processor events, angles life of multi-issue and OoO pipeline processors
Functional Coverage of RISC-V Instructions: Scope

- There are many different instructions in the RV64 extensions:
  - Integer: 56, Maths: 13, Compressed: 30, FP-Single: 30, FP-Double: 32
  - Vector: 356, Bitmanip: 47, Krypto-scalar: 85
  - P-DSP: 318
  - For RV64 that is 967 instructions...
- Each instruction needs SystemVerilog covergroups and coverpoints
  - 10-40 lines of SystemVerilog for each instruction
  - 10,000-40,000++ lines of code to be written
- Not design or core specific

riscvISACOV Is Automatically Generated SystemVerilog Functional Coverage

- riscvISACOV provides functional coverage of Instructions and operands
- Roadmap includes CSRs and data hazards
- Imperas tools can automatically generate functional coverage code for custom instructions

Test Stimuli

- Instruction Stream Generator (ISG) and/or directed tests
- ISG generates test programs using constrained random approach
  - Most often uses the ISG
  - Commercial such as Valtrix STING
  - Open source such as Google riscv-dv
  - Require toolchains like GCC, LLVM for assemblers, linkers
  - Require functional coverage so that you know what you've tested!
- Directed tests
  - Imperas have developed a directed RISC-V test generator, instruction coverage verification IP and a mutating fault simulator (for test qualification) to provide high quality test suites
  - The generated tests suites are targeting architectural compatibility as defined in the Rv64 architectural test working group coverage requirements
  - Free Imperas architectural validation test suites (50+), including RV32/64 I, M, C, F, D, B, K, V, P
  - https://github.com/riscv-ovpsim/imperas-riscv-tests
  - Imperas commercial directed test suites for vector extension, protected memory components
  - Can support any RISC-V vector or PMP configuration; the user selects the configuration and Imperas generates the test suite
**ImperasDV: Debug with MPD**

- Imperas MPD is an Eclipse based debug tool
- Can debug using source line or instruction level
- See new custom instructions and any new additional state registers
- Break at the first mismatch, debug SW and RTL concurrently

**Software Debug and Analysis Tools Automatically Work With the Custom Instructions**

- New custom instructions in trace disassembly
- New custom instructions, new additional state registers

**Hybrid Simulation-Emulation for HW-SW Co-Verification**

- It takes more than just the RTL simulator for comprehensive processor verification
- An additional tool the the hardware emulator, e.g. Cadence Palladium
- The interface to Palladium is via the Cadence Helium SystemC simulator

**Agenda**

- RISC-V and processor verification
- RISC-V processor models
- RISC-V processor verification methodology
- Processor verification success
- Summary
RISC-V Processor DV Results

- OpenHW Core-V-Verif
  - OpenHW users now on 3rd generation of Core-V-Verif DV flow
  - CV32E40P successfully taped out
  - CV32E40Pv2, CV32E40S, CV32E40X, CV32E20 using this flow today; all expected to complete DV this year
- Other successful DV projects using Imperas include Codasip, MIPS, Nagra, NSITEXE, Nvidia Networking, ...

Case Study: Wally RISC-V Core

- Configurable core:
  - RV32I, RV32E, RV64I, RV64E
  - A, C, F, D, M extensions, privileged modes, CSRs
  - MMU/TLB virtual memory, caches
- Developed at Harvey Mudd College / Oklahoma State University
  - Focus: high quality core for processor architecture education
  - Now in OpenHW as CORE-V Wally (https://github.com/openhwgroup/cvw)

- Status in January 2023 – before starting to use RVVI + ImperasDV for verification:
  - Passing all RISC-V International compliance tests, Imperas compatibility tests
  - Passing all RISC-V International compliance tests, Imperas compatibility tests
  - Using Compliance Level post-simulation signature file compare
  - Boots Linux

Wally + RVVI – Status (July 2023)

- RVVI Tracer + testbench integration: 3 days of effort
- Results:
  - 20+ bugs found in simulation almost immediately using ImperasDV and riscv-dv
  - Reached Linux prompt with continuous checking: 2 days of simulation
  - One bug found just after the Linux command prompt (!)
- Functional coverage achieved by booting Linux: covergroups 37%
- Future work:
  - Boot Linux with co-sim using hardware
  - Achieve 100% functional coverage using constrained-random tests

Agenda

- RISC-V and processor verification
- RISC-V processor models
- RISC-V processor verification methodology
- Processor verification success
- Summary
Processor Verification: The Key to the RISC-V Castle

- Asynchronous step-compare methodology
- High quality reference models
- Verification IP
- Verification standards
- Verification metrics
- ImperasDV

Thank you
Larry Lapides
LarryL@imperas.com
Adnan Hamid
Breker Verification Systems
Founder and CTO

Advanced RISC-V Verification Technique Learnings for SoC Validation

Gold Sponsor

Abstract
The verification of application-level RISC-V cores require specialized techniques and approaches previously the purview of Arm, Intel and other processor companies. The open and customizable RISC-V cores have led to many new processor development teams with unique microarchitectural approaches that require extensive verification.

Breker has found that a key aspect of RISC-V core verification involves its smooth operation within the larger system. For example, load-store anomalies, asynchronous interrupt mechanisms, and security protocols are just a few of many issues that must be fully analysed. In developing new test approaches for these and other scenarios, their application in more general System-on-Chips has become apparent, and indeed these methods can track complex system corner cases that will never be detected simply by running real workloads or benchmarks. This presentation will describe many techniques useful for RISC-V core verification, and also how they may be applied to the broader SoC at large for high coverage verification.

Biography
Adnan is the founder and CTO of Breker and the inventor of its core technology. Noted as the father of Portable Stimulus, he has over 20 years of experience in functional verification automation, much of it spent working in this domain.

Prior to Breker, he managed AMD’s System Logic Division, and also led their verification team to create the first test case generator providing 100% coverage for an x86-class microprocessor. In addition, Adnan spent several years at Cadence Design Systems and served as the subject matter expert in system-level verification, developing solutions for Texas Instruments, Siemens/Infineon, Motorola/Freescale, and General Motors.

Adnan holds twelve patents in test case generation and synthesis. He received BS degrees in Electrical Engineering and Computer Science from Princeton University, and an MBA from the University of Texas at Austin.
SoC Integrity
SystemVIP
High Coverage, Automated Apps

IP, End-to-End Test Content
Subsystem Simulation
SystemUVML Scalable, Portable Test Content

RISC-V Core & SoC Integrity
SoC Simulation or Emulation
Comprehensive Toolkits for Stringent Testing

System & SW Validation
Prototyping/Post-Silicon
Firmware Test, Performance Profiling

RISC-V Core SystemVIP

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Instructions</td>
<td>Do instructions yield correct results</td>
</tr>
<tr>
<td>Register/Register Hazards</td>
<td>Pipeline perturbations due to register conflicts</td>
</tr>
<tr>
<td>Load/Store Integrity</td>
<td>Memory conflict patterns</td>
</tr>
<tr>
<td>Conditionals and Branches</td>
<td>Pipeline perturbations from synchronous PC change</td>
</tr>
<tr>
<td>Exceptions</td>
<td>Jumping to and returning from ISR</td>
</tr>
<tr>
<td>Asynchronous Interrupts</td>
<td>Pipeline perturbations from asynchronous PC change</td>
</tr>
<tr>
<td>Privilege Level Switching</td>
<td>Context switching</td>
</tr>
<tr>
<td>Core Security</td>
<td>Register and Memory protection by privilege level</td>
</tr>
<tr>
<td>Core Paging/MMU</td>
<td>Memory virtualization and TLB operation</td>
</tr>
<tr>
<td>Sleep/Wakeup</td>
<td>State retention across WiFi</td>
</tr>
<tr>
<td>Voltage/Freq Scaling</td>
<td>Operation at different clock ratios</td>
</tr>
<tr>
<td>Core Coherency</td>
<td>Caches, evictions and snoops</td>
</tr>
</tbody>
</table>

SoC SystemVIP

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Memory Tests</td>
<td>Test Cores/Fabrics/Memory controllers</td>
</tr>
<tr>
<td>Random Register Tests</td>
<td>Read/write test to all uncore registers</td>
</tr>
<tr>
<td>System Interrupts</td>
<td>Randomized interrupts through CLINT</td>
</tr>
<tr>
<td>Multi-core Execution</td>
<td>Concurrent operations on fabric and memory</td>
</tr>
<tr>
<td>Memory Ordering</td>
<td>For weakly order memory protocols</td>
</tr>
<tr>
<td>Atomic Operation</td>
<td>Across all memory types</td>
</tr>
<tr>
<td>System Coherency</td>
<td>Cover all cache transitions, evictions, snoops</td>
</tr>
<tr>
<td>System Paging/IOMMU</td>
<td>System memory virtualization</td>
</tr>
<tr>
<td>System Security</td>
<td>Register and Memory protection across system</td>
</tr>
<tr>
<td>Power Management</td>
<td>System wide sleep/wakeup and voltage/freq scaling</td>
</tr>
<tr>
<td>Packet Generation</td>
<td>Generating networking packets for I/O testing</td>
</tr>
<tr>
<td>Interface Testing</td>
<td>Analyzing coherent interfaces including CXL &amp; UCie</td>
</tr>
<tr>
<td>SoC Profiling</td>
<td>Layering concurrent tests to check operation under stress</td>
</tr>
<tr>
<td>Firmware-First</td>
<td>Executing SW on block or sub-system without processor</td>
</tr>
</tbody>
</table>
Agenda

- Test Suite Synthesis and SystemVIP
- RISC-V Core Verification SystemVIP
- RISC-V SoC Verification SystemVIP

The High Cost of Developing Test Content

- Largest Functional Verification Challenge
- Verification: Content Development 30%
- Debug: 25%
- Verification: Other 13%
- Design: 32%

Why? Resource Intensive Test Content

- Complex tests hard to get right
- Why? Resource Intensive Test Content
- Can we abstract verification intent?
Requires automation, reuse and other new thinking.

Arm spends $150M per year on 10 verification cycles per core.

Library Hard for RISC

Lots of applications expands verification requirements.

The fabrics Silicon

Open Instruction Set Architecture (ISA) creating a discontinuity in the market

Appears to be gaining significant traction in multiple applications

Significant verification challenges

- Arm spends $150M per year on 10 verification cycles per core
- Hard for RISC-V development group to achieve this same quality
- Lots of applications expands verification requirements
- Requires automation, reuse and other new thinking
Different Challenges for Core vs SoC Verification

RISC-V Core Verification Challenges
- System Security
- System Synchronization
- Coherence
- Power Management
- Memory Management
- Interrupts
- Atomic Operations
- Random Instructions
- Load/Store Integrity
- Context Switching
- Pipeline Perturbations
- Memory Conflict Patterns
- Ocram, Flash
- System Memory Virtualization
- Cover All Cache Transitions, Evictions, Snoops
- System Security
- System Synchronization
- Coherence
- Power Management
- Memory Management
- Interrupts
- Atomic Operations
- Random Instructions
- Load/Store Integrity
- Context Switching
- Pipeline Perturbations
- Memory Conflict Patterns
- Ocram, Flash
- System Memory Virtualization

RISC-V SoC Verification Challenges
- System Security
- System Synchronization
- Coherence
- Power Management
- Memory Management
- Interrupts
- Atomic Operations
- Random Instructions
- Load/Store Integrity
- Context Switching
- Pipeline Perturbations
- Memory Conflict Patterns
- Ocram, Flash
- System Memory Virtualization
- Cover All Cache Transitions, Evictions, Snoops
- System Security
- System Synchronization
- Coherence
- Power Management
- Memory Management
- Interrupts
- Atomic Operations
- Random Instructions
- Load/Store Integrity
- Context Switching
- Pipeline Perturbations
- Memory Conflict Patterns
- Ocram, Flash
- System Memory Virtualization

Agenda
- Test Suite Synthesis and SystemVIP
- RISC-V Core Verification SystemVIP
- RISC-V SoC Verification SystemVIP

RISC-V Core Testbench Integration

RV64 Core Instruction Generation

Instruction Coverage Analysis
**Example Address Allocation Patterns**

- Random Clusters with locality of reference
- Stride Patterns across fixed address distances
- Sequential Addresses matching a specific Hash

**Application to Unit Bench and Sub-System Bench**

**RV64 Core Exception Testing**

Generates for example, `asm("UNIMP");`
Check exception counts
Page Based Virtual Memory Tests

RV64 Core Page Based MMU Tests

Swap MMU PTE's and Check memory access

Agenda
- Test Suite Synthesis and SystemVIP
- RISC-V Core Verification SystemVIP
- RISC-V SoC Verification SystemVIP

RISC-V SoC Testbench Integration
Multi-Agent Scheduling Plans: Overview

- True Sharing within scenario
- False Sharing across scenarios

RV64 MultiCore MoesiStates

Efficacy of System-Integrity Testing using the RISC-V TrekApp

Atomics Testing
● Assume initial state A=0, B=0

● The Dekker Algorithm States

  core 0: ST A, 1; MEM_BARRIER; LD B
  core 1: ST B, 1; MEM_BARRIER; LD A

error iff \( (A == 0 \&\& B == 0) \)

● This is a test for a weakly ordered memory system

  o Such a system must preserve the property that a LD may not reorder ahead of a previous ST from the same agent

Dekker Memory Ordering

MultiCore MMU Tests

False-Share Memory Stress Tests
Thanks for Listening!
Any Questions?
Notes
Abstract
Simulation is the most common RTL verification technique, involving the execution of testbenches are essential for the verification of the designs. SystemVerilog and UVM have been widely adopted over the last two decades. However, the complex nature of these languages/methodologies can make it difficult for junior-level engineers to create maintainable and reusable code.
Static linting checks have been widely used for RTL design, but they have not been used as widely adopted for Testbenches. In this talk, we share our experience in using a popular open-source framework named PySlint to lint-check SystemVerilog UVM Testbenches. We show how PySlint can be used to identify potential problems in SV-UVM Testbenches, such as coding style violations, potential bugs, and potential performance bottlenecks. We also show how PySlint can be used to generate reports that can help engineers to improve the quality of their Testbenches building a robust verification environment process. Some key components of a robust verification environment includes, Testbenches, coverage matrices, Assertions, regression testing.
We believe that PySlint can be a valuable tool for improving the quality of SystemVerilog UVM testbenches. By using PySlint, engineers can identify and fix potential problems in their testbenches early in the development process, which can help to prevent costly delays and errors.

SystemVerilog:
Biography
15+ years of Design Verification exp.
Prior to Broadcom, I worked for Microchip, Western Digital.
My Verification expertise from IP level to SOC subsystems
Initially, I worked for VIP development, Net works chips, Mobile chips, and Storage product lines, Last 10 years mainly with Wireless products, GPS and Bluetooth, Wifi SoC's.
Lint and Code Analysis

- Find bugs early
- RTL Lint well proven
- Verification
  - SystemVerilog TB
  - UVM Code base
  - Assertions
  - Acceleration
  - Emulation
  - Formal Verification

Source: Siemens EDA

RTL Lint is Very Popular

- RTL linting is widely used in the field of digital design
- It helps to identify and fix issues in RTL code
- Ensures adherence to naming guidelines and coding style checks
- Improves overall code quality and reliability

Open-source RTL Lint - options

- Widely used flow - RTL Lint
  - Naming guidelines
  - Coding style checks
  - Synthesis checks
  - DFT checks
- Many teams now integrate this with CI/CD flows
- Good opensource alternatives becoming available
TB Lint - Challenges and Alternatives

- New Paradigm: Code analysis approaches changing
- Lack of good parsers/tools/API for newer languages
- Slang Verible, Slint, and PySlint are good opensource alternatives
- Provide flexibility and customization options

Testbench Linting

- New paradigm
  - Old/proven concept
  - Lack of good parsers/tools/API
- Verible – C++ based rules
  - https://github.com/chipsalliance/verible
- Slint – Rust based, custom plugin
  - https://github.com/dalance/svlint
- UVMLint – Python based, custom plugin for UVM TB
  - https://github.com/AsFigo/UVMLint
- PySlint – Python based, works on top of slang/pyslang pkg
  - https://github.com/AsFigo/pyslint

Testbench Linting – Sample rules

<table>
<thead>
<tr>
<th>PySlint Rule ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME_CE_MISMATCH</td>
<td>Naming/style checks</td>
</tr>
<tr>
<td>NAME_CE_MISMATCH</td>
<td>Assertion specific, Performance, Debug categories</td>
</tr>
<tr>
<td>UVM_VIOLATION</td>
<td>Class specific - style, Functionality</td>
</tr>
</tbody>
</table>

SV Constraints – case study

```
class bug_ce:
    rand bit num_list[32];
    constr_from c_sum (num_list.num) = 1;
endclass
```
PySlint – results

UVMLint – results

Testbench Linting - summary

• Exciting new frontiers
• Open-source can now lead the way for commercial tools to follow
  – Than just playing a catch-up game
  – Faster turn-around time
  – Potentially vast set of developers!

Summary, next steps

• Open-source verification is now real
• Shared our experience in:
  – Assertions
  – Unit Testing
  – RTL Linting
  – Testbench Linting
• PySlint – a case where open-source can leapfrog beyond commercial tools
  – Opportunity for US Universities to take the lead!
Our contributions – GitHub repos/projects

Who we are, what As-Figo do?

• UK based R&D design start-up
  • Bringing decades of Chip Design and EDA expertise
  • Globally reputed training solutions
  • RTL Design, Verification
    • SystemVerilog, VHDL, etc.
    • Methodologies: UVM, OVM, VMM, etc.
  • Bringing AI/ML to chip design flows
    • Generative AI
    • AI Analytics

Summary, next steps

• Proved that open-source is now ready for primetime in Design-Verification
• All our code will be available as open-source via GitHub - https://github.com/svenka3/as-fipo-verilator
• Looking for volunteers to fix issues on Verilator, improve SVA support
  - svenka3@gmail.com
  - Balram.Meghavath@broadcom.com

Integrate Lint to CI/CD

• Subset of lint rules
• Customize as per project stage/phase
• Stricter as it gets mature
• Applies to RTL & TIL
• Ideally use open-source tools
• Free of license restrictions
• Easy to customize
• Commercial license models available (MIT)
Conclusion

Integration with CI/CD flows helps in continuous improvement

Adopting new approaches and alternatives is essential for evolving languages and paradigms

RTL & TB linting and code analysis are essential for quality digital design

Using popular tools and alternatives improve code reliability

Conclusion
Notes
Abstract
With all the innovations in tools to help with functional closure, one that is not fully appreciated is the need for disciplined documentation at every stage of verification. Documentation often is a victim of tight schedules and yet lack of it remains the root cause of schedule slips, many escapes, delays and churn in projects. We will explore different kinds of documents and shine light on this problem. We claim, without due appreciation of this, we can not close the verification gap.

Biography
Hemendra Talesara is a distinguished leader in the semiconductor sector in the USA, amassing a rich legacy of over 35 years. He served in senior management roles with IBM, AMD, Synopsys, and many other organizations. Specializing in CPU, GPU, ASIC, and SOC chip design, he boasts profound chip design verification technology expertise. Hemendra's exceptional competencies encompass global team formation, project risk management, and certified corporate governance as a NACD Certified Corporate Board Director. His interests include business Implications of digital disruption, AI, and cyber security. Hemendra's academic journey encompasses an MS in Electrical and Computer Engineering from the University of Texas at Austin and a BE in Electronics and Telecommunication Engineering from IIEST, Shibpur, India. Beyond his professional journey, Hemendra's active involvement in industry conferences, technical journals, teaching, travel, and diverse hobbies showcases his multifaceted persona. With a career defined by inclusive and collaborative leadership, technical prowess, and a commitment to innovation, Hemendra Talesara brings considerable depth to any organization.
Notes
SELECT SOURCES OF BUG ESCAPES

1. REQUIREMENT DOCUMENTS
   ERRORS, OVERLOOKS OR GAPS IN THE REQUIREMENTS
2. SPECIFICATIONS
   ERRORS IN THE DESIGN OR ARCHITECTURE. MISCREATING ASSUMPTIONS
3. IMPLEMENTATION
   ERRORS IN THE CODING OR IMPLEMENTATION.
4. VERIFICATION PLAN
   ERRORS IN THE TEST PLANNING OR TEST ACTIVITIES.
5. COVERAGE PLAN
   GAPS IN TESTING. VERIFICATION UNIVERSE IS VERY LARGE
6. REVIEWS (DOCUMENTS AND CODE)
   ERRORS IN THE PROCESS OR POLICIES
BUGS IN REQUIREMENT DOCUMENTS

ERRORS, OVERSIGHTS OR GAPS IN THE REQUIREMENTS
• A REQUIREMENT IS OMITTED OR FORGOTTEN
• PHRASED POORLY
• NOT PROPERLY UNDERSTOOD BY ARCHITECTS
• MISUNDERSTOOD BY DESIGNERS

BUGS IN SPECIFICATIONS

ERRORS IN THE DESIGN OR ARCHITECTURE
• ARCHITECTS / DESIGNERS CREATE AN INEFFICIENT ALGORITHM OR CONFIGURATION
• ALGORITHM OR CONFIGURATION DOESN'T YIELD THE REQUIRED PERFORMANCE OR POWER
• SPECS PHRASED POORLY OR INADEQUATELY
• SPECS MISUNDERSTOOD BY ENGINEERS
• LEGACY SPECS MISSING OR INADEQUATE

BUGS IN IMPLEMENTATION

ERRORS IN THE CODING OR IMPLEMENTATION
1. TRADITIONAL BUGS FROM SIMPLE TO COMPLEX
2. UNGRACEFUL ERROR HANDLING
3. LACK OF DOCUMENTATION IN CODE IMPLEMENTATION
4. LEGACY CODE MISMATCH WITH ARCHITECTURE UPDATES
5. INADEQUATE DESIGN SPEC

GAPS IN VERIFICATION PLAN

ERRORS IN THE TEST PLANNING OR TEST ACTIVITIES
1. RIGHT METHODOLOGY, RIGHT TESTBENCH ARCHITECTURE
2. GAPS IN SCENARIO PLANNING
3. INCOMPLETE BOUNDARY CONDITIONS
4. CONSTRAINT SETTING, TWEAKING
5. COVERAGE PLAN AND FEATURE TESTING TRACEABILITY TO SPECS
6. TRACKABLE ENUMERATED TEST TEMPLATES
GAPS IN COVERAGE PLAN

INADEQUATE TESTING DUE TO FALSE CONFIDENCE
1. VERIFICATION UNIVERSE IS VERY LARGE
2. PSEUDO-RANDOM TESTING NEEDS TO BE CONSTRAINED
3. GAPS IN PLAN CAN LEAD TO FALSE CONFIDENCE AND INADEQUATE TESTING
4. COVERAGE PLAN NEEDS TO BE REVIEWED CAREFULLY

NO TIME FOR REVIEWS

ERRORS IN THE PROCESS OR POLICIES
1. REVIEWS CONSIDERED SIDE ACTIVITY (NOT A CENTRAL CONTRIBUTION)
2. APPROVALS WITHOUT ADEQUATE REVIEWS OF DESIGN, CODING OR TESTING.
3. REVIEWING RESOURCES INCLUDE TEAMS WITH OTHER PRIORITIES
4. TIME/RESOURCE ALLOCATION FOR REVIEWS ARE DIFFICULT TO PLAN AND YET ARE MOST CRITICAL

TAKEAWAYS
1. VERIFICATION IS A “WRITTEN” CONTRACT
2. GAPS IN DOCUMENTS = GAPS IN VERIFICATION
3. ROI ON DOCUMENTATION CLOSURE
   ✓ IMPROVED QUALITY
   ✓ IMPROVED SCHEDULE (COUNTER INTUITIVE?)
   ✓ IMPROVED MAINTENANCE

CONTACT US
HEMENDRA.TALESARA@GMAIL.COM
MOBILE: (512) 657-7520
We would be grateful if you could move to the track session as quickly as possible.
Notes
Aditya Devarakonda

NXP Semiconductor
Senior Manager – Design Verification

Leveraging AMS verification and DMS verification for efficiency and quality in Mixed-signal designs

User paper

Abstract
Mixed-signal verification at the SOC level is a unique problem that is currently being tackled by two disciplines of verification- Analog Mixed-Signal (AMS) Verification, and Digital Mixed-Signal Verification (DMS). Each of these disciplines have their own strengths, and weaknesses. Understanding and leveraging those appropriately is needed to achieve the required success in silicon, while using the verification engineering resources efficiently. In this paper I would like to discuss the aspects that need to be considered while determining an effective, and combined AMS and DMS strategy for robust verification of Mixed-signal designs.

Biography
Aditya Devarakonda leads Digital Verification for Advanced Power Systems at NXP. Prior to this he held several roles in digital design and digital verification of Mixed-signal ICs at ON, Maxim (ADI), Dialog (Renesas), and Freescale. Aditya holds an MSEE.
Notes
AGENDA
What is a Mixed-Signal IC?
Verification of Mixed-Signal ICs
Evolution of DV into DMS
Need for a combined DMS-AMS strategy
A combined AMS-DMS flow
Transitioning from an AMS-DV to AMS-DMS flow
More on Modeling
Combined AMS-DMS flow – Some Pitfalls
Conclusions
**Verification of a Mixed-Signal IC**

- Analog-Mixed-Signal Verification (AMSV)
  - SOC level verification with Analog focus
  - Analog is typically represented using schematics/models that need the simulator’s analog solver
  - Digital is typically represented using behavioral model (Verilog/SV/VHDL etc)
  - A more accurate verification method that is orders of magnitude slower than DMSV
- Digital Verification (DV)
  - SOC level (?) verification with digital focus
  - Analog is typically represented using behavioral models that do not need the simulator’s analog solver
  - Digital is typically represented using behavioral model (Verilog/SV/VHDL etc)
  - Orders of magnitude faster than an AMS simulation
  - Accuracy depends on modeling, and model verification

**Evolution of DV into DMS**

- Evolution of digital verification in Mixed-Signal ICs
  - Dig-top only verification
    - Verification of the digital was limited to the digital top with no true SOC level verification
    - Lack of modeling of any analog blocks
    - Analog response is provided by the testbench itself
    - 100% digital only focused verification flow
    - SOC level connectivity and functional coverage are heavily dependent on AMS
  - Dig-top with functionally equivalent Analog models
    - Analog models functionally equivalent to the analog schematics used along with digital top
    - No netlisting from the actual chip level schematic
    - Still a very digital only focused verification flow
    - Can result in more accurate digital but SOC level connectivity and functional coverage are still heavily dependent on AMS
  - Dig-top with accurate Analog models and netlisting from chip level schematics
    - SOC level Digital-Mixed-Signal (DMS) verification
    - More accurate and fast Analog behavioral models are used through Verilog, System Verilog Real Number (SV-RNM), and User Defined Net
    - When combined with robust model verification, can provide reliable SOC level connectivity and functional coverage

**Need for a Combined AMS-DMS Flow**

- AMSV and DMSV flows have the same common goal. They solve the same problem
- With more accurate modeling, SOC level use cases and scenarios previously verified only using AMSV could be verified using DMSV without invoking the analog solver
- Concepts like randomization, coverage etc could be realized at the SOC level
- Though DMSV could provide a greater share of functional coverage in less time it still cannot completely replace AMSV
- AMSV and DMSV working independently is inefficient, and unnecessarily redundant
- A combined DMS-AMS strategy which leverages on strengths of DMSV, and AMSV flows will reduce cycle times while ensuring quality

**A Combined AMS-DMS Flow**

- This flow aims to move away from heavy dependence on AMSV for SOC level verification
- Verification Planning
  - Instead of working on separate, and independent Verification Plans, AMSV and DMSV execute a Common Verification Plan
  - Verification Plans clearly identify the method to be used to cover each feature – AMSV, DMSV, or Both
  - The classification depends on the following factors:
    - Criticality of the feature – input from product owners, analog designers
    - Stability of the analog IP and the analog models
    - Existence of a robust analog model/schematic verification flow
    - Other factors like analog, and digital design approach, experience of the teams greatly affect the division of verification coverage between AMSV and DMSV.
  - AMSV and DMSV leads need to be clear on the limitations and strengths of each flow in verifying each feature
Some pitfalls in the suggested approach also have been presented based on some real case studies.

**An AMSV heavy approach with very complex analog models**

- Design teams need to adopt a more top-down design approach
- Define the chip top level pins, architecture, sub-blocks
- Define sub-block level, and Analog/Digital interactions clearly before design implementation
- Well-defined partitioning of analog, and digital blocks at chip, and sub-block level
- A common testbench for AMS and DV is desirable
- Some of the tools allow for the development of a common TB for AMS and DMS
- The test bench configuration determines if the analog solver is invoked or not in a given sim; hence by allowing the same test to be run using AMS or DMS

- Modeling and model verification
- The integrity and reliability of DMSV depends on the availability of accurate and well-written analog models
- Analog models should be thoroughly verified against schematics
- The Model Vs Schematic regression should be run whenever the design changes

**Analog IP development requirements could be extended to include analog model for each IP**

- Given the iterative nature of analog design, this needed the models to be updated too often

- **Model Vs Schematic Verification Flow**
- Given the longer simulation times for AMSV, this approach results in very large verification cycle times
- AMSV effort becomes the bottleneck for top-down

**Case 1** – An AMSV heavy approach with very simple analog models
- Given the longer simulation times for AMSV, this approach results in very large verification cycle times
- AMSV effort becomes the bottleneck for top-down

**Case 2** – A DMSV heavy approach with very complex analog models
- In this approach even minute features like analog trims are modeled
- Given the iterative nature of analog design, this needed the models to be updated too often
- DMSV effort becomes the bottleneck as model failures delay regression closure

**Case 3** – An AMSV-DMSV flow without a strong top-down design flow
- Without a strong top-down design flow, analog schematics, sub-block ports, and the analog-digital interface change frequently requiring frequent netlist, and testbench debug
- DMSV team spends a lot of time in re-netlisting the design, and in testbench debug

**COMBINED AMS-DMS FLOW – SOME PITFALLS**

**Conclusions**
- A combined approach for SOC level functional verification of mixed-signal designs has been presented
- Adopting a well-balanced, and combined AMS-DMS verification flow will greatly reduce total time and resources spent on verification
- Having a robust analog model creation, and verification flow will ensure reliability in the coverage achieved through DMS
- Some pitfalls in the suggested approach also have been presented based on some real case studies
Abstract
The SigmaSense touchscreen controller IP uses proprietary configurable DSP logic for touchscreen signal generation and detection. MATLAB modelling is first used to refine the design to the point of a bit-width accurate resolution of each point in the signal path. This detailed definition is used to define the rtl implementation requirements. Via the MATHWORKS DPIGEN utility an equivalent C model is produced to serve as the DV predictor. A UVM testbench is created where constrained random simulations are performed, comparing the DUT and MATLAB C model results until they are in agreement.

Biography
Bill Tiffany has an extensive background in digital logic design and verification including DSP applications, networking interfaces, microcontrollers, solid state drive controllers and with SigmaSense’s latest touch screen controller. In addition to understanding system requirements, continuous learning and sharing within a team is key to success.
Notes
Both MATLAB and SIMULINK offer DPI C support

- MATLAB (used by our system architects)
- SIMULINK (used by our software architects)

The UVM testbench has to deal with these flattened arrays,

- MATLAB DPIGEN places syntax restrictions on the MATLAB. Legacy code may have to be updated.
- Handle the array element accessing via complex indexing
- Convert the flattened arrays back to n-dimensional arrays using MATHWORKS Porting_DPIC.mk script
- Unzip in Linux sim environment and run using MATLAB C model example
- Seed from SV can be passed as input arg to stimulus model for internal randomization

MATLAB C model example:

- MATLAB DPIGEN produces.zip file with .c, .h, and .sv files
- Unzip in Linux sim environment and run using MATLAB C model example
- Generates library file is generated

In testbench
- Filelist.m
- firFilter_predictor.sv
- firFilter_dpi_pkg.sv
- Methodology:
  - In a given simulation the CSR settings will be set via constrained randomization and applied to DUT, Stimulus and Predictor models
  - Seed from SV can be passed as input arg to stimulus model for internal randomization

MATLABlsiC model example:

- MATLAB C model example:
  - Produces.zip file with .c, .h, and .sv files
  - Unzip in Linux sim environment and run using MATLAB C model example
  - No library file is generated

In testbench
- Filelist.m
- firFilter_predictor.sv
- firFilter_dpi_pkg.sv
- Methodology:
  - In a given simulation the CSR settings will be set via constrained randomization and applied to DUT, Stimulus and Predictor models
  - Seed from SV can be passed as input arg to stimulus model for internal randomization

MATLAB DPIGEN supports export of C models that can be integrated into a SV TB via DPI interface. It supports 1D, 2D, and 3D arrays. Variable size multi-dimensional arrays that are MATLAB function i/o arguments must be flattened in MATLAB before passing to DPIGEN.

Sigm aSense methodology:

- MATLAB is used to refine the system architecture for maximum performance at minimum gate count.
- MATLAB model is refined to the point of being bit width accurate so any distortion effects are understood.
- RTL design requirements are extracted from the detailed MATLAB models.
- Design Verification
  - Design engineer can do initial debug with testvector data files exported from MATLAB sims and used as stimulus/checking in a simple SV testbench.
  - DPI C models from DPIGEN are used by DV team for stimulus and predictors.
  - In a given simulation the CSR settings will be set via constrained randomization and applied to DUT Stimulus and Predictor models.
  - MATLAB stimulus C models emulate Rx data from the touchscreen.
  - Seed from SV can be passed as input arg to stimulus model for internal randomization.
MATLAB C model example:

- In simulation command line
  - xrun –dpi –sv_root ../cpred/lib_firFilter –sv_lib lib_firfilter.so

- NOTES:
  - "double" datatype in MATLAB args translate to "real" datatype in SV
  - Input args of C function must be "real" so conversion between "logic" and "real" types is required. Recommend hiding these MATLAB specifics in "predictor" so "scoreboard" can be more generic.
  - MATLAB multi-dimension variable size arrays must be flattened when they appear as I/O arguments
  - Predictor C model will produce an array of expected data for entire simulation
  - Scoreboard will check DUT output cycle by cycle, updating index into predictor array on each cycle.
  - Similarly, a driver will index through a stimulus array cycle by cycle.
  - Matlab model should provide enough intermediate datapath values to isolate an rtl/matlab mismatch

MATLAB C model challenges:

- Keeping track of indexing into flattened multidimensional variable size arrays can be confusing.
  - DV engineer needs to understand the matlab internal array structure before flattening and how flattening was done (columns major vs row major ordering)
- Some matlab may need to be recoded to live within codegen restrictions
  - Add %#codegen comment to matlab function, matlab will indicate coding issues
- Agreement between designer and matlab creator on module partitioning, interfaces and important internal signals to be compared is important
Abstract
UVM testbench, environment and UVC development practices need a boost. UVMGen speeds VIP creation by reactively generating code that uses only the best industry strategies. Now, recent college graduates can create environments that will withstand any guru’s code review and they’ll be running tests in a matter of hours, not months. At integration levels, simply click in these lower level environments to create cluster and chip level testbenches. UVMGen ensures seamless compatibility for UVC and environment reuse, making development and integration a snap.

Biography
Ben Delsol has been a DV engineer with a passion for improving quality of work with automation. For more than 15 years, Ben has been a part of creating and observing industry best practices at companies such as Intel, Qualcomm, Samsung and Microsoft. He leaves his mark at these companies creating tools that transform the way work is done. Now Ben has started his own company, UVMGen LLC, which is positioned to change the way the world does DV. Never has a SystemVerilog code generation tool been created with this degree of intelligence and Ben is excited to introduce it here at the Tessolve DV Conference in Austin.
Intro

- Who am I?
  - Ben Delsol - DV engineer formerly at Intel, Qualcomm, Samsung and Microsoft.
  - Founder of uvmgen.com.
- What I care about?
  - Clean code.
  - Methodology best practices.
  - Not wasting brain energy.
    - Divide, reuse and conquer.
    - Automating redundant problems.

The idea of UVM is spot on

- Common procedures and methodologies across the industry.
- Clear coding and separation of testbench concerns.
- Reusable protocol agents.
- Reusable block level environments.
- Decades of verification best practices rolled into one methodology.

Some best practices from the last 15 years…

- Pass down config object over config db
- Use slave sequences with late response randomization
- Reset methodology: don’t kill sequences with the sequencer
- Use virtual sequences over phase jumping
- No virtual sequencers
- Use objections wisely
- Use constraint policies over inheritance
- DUT parameter passing to VIP
- Interface harnesses
- Abstract/concrete classes
- Use sequence, BFM and config factory overrides
- Scale UVC, sub-env, config and TLM instances at runtime
- Conditional instantiation of static verification elements at compile time
- Use standalone testbench for UVC development
- And many more…
Abstract/concrete classes

- **Problem:**
  - Any component which uses a virtual interface handle to a parameterized interface must be parameterized, and so too must all its component ancestors (i.e., test, env, agents, drivers, monitors all must be parameterized! Possibly configs, sequencers and sequences too).

- **Solution:**
  - Define a BFM class in the parameterized interface.
  - Have the component initialize the BFMs construction with the abstract/concrete design pattern.
  - Retain the BFMs access to its config object, UVM printing and ability to be overridden by the factory.

- Can be used for access of protocol checkers and signal checkers too.

DUT parameter passing to VIP

- **Problem:**
  - The verification environment of a parameterized design must also have access to those parameters. Same problem as getting access to a parameterized virtual interface handle, type specialization of many classes can become very cumbersome to manage.

- **Solution:**
  - In your interfaces, collect parameter values in an object and put the object in the uvm_config_db.
  - In your configuration, get the object with parameter values out of the uvm_config_db.
  - In your interface harness, collect DUT parameter values in an object and put them in the uvm_config_db.
  - In your env-config, get the object with parameter values out of the uvm_config_db.

Use constraint policies over inheritance

- **Problem:**
  - Mixing and matching constraints via inheritance causes a lot of copy and paste/repeat of constraint code. How to DRY up my code (i.e., not repeat myself)?

- **Solution:**
  - Write your constraint once in a policy object. Then apply the policy objects on the sequence item, sequence or config object as needed.
  - In your test, factory override sequences, configs with the new classes that apply these policies.

Interface harness

- **Problem:**
  - Code which handles connectivity of the design to interfaces, access to BFMs and DUT parameters is not reusable from the block level to upper levels of integration.

- **Solution:**
  - An interface harness defines the connectivity of all interface signals to design signals and can be reused bound into the DUT at block level as well as upper levels of integration.
  - It encapsulates access to BFM concrete creation classes through the config db.
  - It encapsulates collecting and setting DUT parameters in the config db.
No virtual sequencers - pass sequencers to the vseq

- **Problem:**
  - How to get sequencers and configuration objects to virtual sequences and their sequences in a simple yet reusable way? i.e., reuse the virtual sequence in upper levels of integration where its environment and virtual sequencer might not be instantiated.

- **Solution:**
  - Set sequencers and configuration objects with virtual sequence setter functions.
  - Define a `set_sequencers(virtual_sequence_base vseq)` function in the base test which assigns agent sequencer handles to the top virtual sequence.
  - Define a `set_sequencers(virtual_sequence_base vseq)` in the `virtual_sequence_base` class which assigns agent sequencer handles to lower level virtual sequences.

Scale UVC, sub-env, config and TLM instances at runtime

- **Problem:**
  - Compile-time instance scaling requires a proliferation of parameter passing via class type specializations.

- **Solution:**
  - Collect DUT parameters at runtime and make available to env and agent configs.
  - Construct agents and sub-env instances as needed.
  - Construct env-config and agent config instances as need.
  - Construct/ connect scoreboard, predictor and coverage TLM as needed.
  - Construct sub-env predictors as needed.

Conditional instantiation of static verification elements at compile time

- **Problem:**
  - Sub-environments and SVA may not be needed in every test regression and can bog down full-chip simulation performance.

- **Solution:**
  - Make instantiation of static verification sub-elements, such as interfaces, protocol checkers and signal checkers, conditional at compile time.
  - Create clear, easy to use macro definitions to disable binding of verification elements individually or all at once.
  - Enable verification sub-environments and/or SVA as needed for debug.

But the UVM dream is not today’s reality

- **Current state:**
  - Tight schedules.
  - A daunting programming effort.
  - Some UVM best practices unknown or time-consuming to implement.

- **The result:**
  - Careful implementation of best practices for reuse and scalability.
  - Get it verified. On time. However possible.
  - Monolithic verification decisions made to hit deadlines.
  - Hacks to fix hacks.
Introducing UVMGen Technology

- With the best DV practices across the industry distilled and encoded into the UVMGen code generator, users can stand on the shoulders of DV experts.
- Generate world-class VIP in an instant, reuse at a click and scale with ease.
- Now everybody can code like a guru and capitalize on the UVM promise letting verification productivity, reuse and confidence shoot through the roof.

uvmgen.com
We would be grateful if you could move to the track session as quickly as possible.
Doug Smith
Doulos
Engineer / Instructor

Using Non-Determinism with Formal

Gold Sponsor

Abstract
The use of non-determinism with formal is how formal is able to manage large state spaces and still arrive at a quick solution. Non-determinism plays a part in writing our formal constraints, formal targets, and formal abstractions. In this formal tutorial session, we'll explain what non-determinism is, how it's used, and show lots of examples so you can take advantage of non-determinism in verifying your designs.

Biography
Doug Smith is a verification engineer and instructor for Doulos based in the Austin Texas area with expertise in UVM and formal technologies. He has been using formal technology for several decades, performing formal verification on many kinds of designs and formal applications. Likewise, he has provided formal application support at both Jasper and Mentor/Siemens EDA. At Mentor/Siemens EDA, he served as a formal specialist and verification consultant, where he provided both formal consulting and developed an automotive functional safety formal app for performing formal fault campaigns. At Doulos, he delivers training in verification methodologies like UVM, SystemVerilog, and formal technology.

Doug holds a masters degree in Computer Engineering from the University of Cincinnati and a bachelors degree in Physics and Biology from Northern Kentucky University. Currently, he resides in Paige Texas with his wife and family on a small farm where he raises bees, cows, horses, chickens, and pigs and loves driving a tractor.
Global Training Solutions

ESL & Verification Methodology
- SystemVerilog » UVM
- SystemC » TLM-2.0 » Formal

Hardware Design (ASIC / FPGA)
- VHDL » Verilog » SystemVerilog
- Tcl » AMD » Intel FPGA

Arm & Embedded Design
- Arm Cortex A/R/M Series » C » C++
- RTOS » Linux » Yocto » Security

AI & Deep Learning
- Edge AI » Deep Learning
- Python

EDA playground
Practice – Share – Learn
Simulate your hardware description code in a web browser for free

Call +1-888-GO-DOULOS to discuss your training needs
www.doulos.com
How nondeterminism accelerates formal verification

- Randomness in formal
- Nondeterminism accelerates formal
- Wrap-up

A Case for Randomness

- Tests what we don’t know
- Faster test development
- Find corner cases faster
- Deeper state space exploration

Not Just for Simulation

Determinism in algorithms is reproducibility of results with the same inputs

Simulation
- Order of process execution is nondeterministic
- Everything else is deterministic - random stability + reproducibility

Formal
- Algorithms are nondeterministic – different results every time!
  i.e., randomness is built into formal by default

Randomness is not just for simulation!
Some Things Aren’t So Different

Randomization in ...

<table>
<thead>
<tr>
<th>Simulation</th>
<th>Formal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write random constraints</td>
<td>Constrain formal’s randomness</td>
</tr>
<tr>
<td>Test what we don’t know</td>
<td>Frees formal to test everything</td>
</tr>
<tr>
<td>Hit corner cases faster</td>
<td>Skips straight to target</td>
</tr>
<tr>
<td>Coverage shows what was hit</td>
<td>Coverage shows formal’s path</td>
</tr>
</tbody>
</table>

But Wait, There’s More

Randomness (nondeterminism) in formal also helps with ...
- Reducing the cone of influence
- Abstracting away complexity
- Simplifying property writing
- Dealing with inconclusives

Nondeterminism in Formal

Anything undriven is a formal control point (random)

DUT inputs
Cut signals
Black-boxed ports
Undriven signals and variables

Cut Points and Black Boxes

Assertions
Cut Points and Black Boxes

Cut Points

Assertions

netlist cutpoint signal

snip_driver signal

stopat signal

Unconstrained "free variables"

Initial Value Abstraction

always @(posedge clock or posedge reset)
if (reset)
    count <= $;
else if (count < 16'hffff)
    count <= count + 1'b1;
else
    count <= $;

assign en1 = (count == 16'h01ff);
assign en2 = (count == 16'h07ff);
assign en3 = (count == 16'h3fff);
always @(posedge clock)
begin
    if (en1)
        array[address1] <= data_in; // Bounds check
    if (en2)
        array[address2] <= data_in; // Bounds check
    if (en3)
        array[address3] <= data_in; // Bounds check
end

With no reset, count is a free variable

Fails at depth=1

Fails at depth=1

Fails at depth=1
Free Variables

Known as random, free, or nondeterminism (ND) variables

Undriven signals and variables are formal control points

```verilog
module formal_tb [ ... ] output_data ];
byte data; // Data output
byte random_data; // Undriven variable => random
always @(posedge clk or negedge rst_n)
  if (!rst_n)
data <= '0;
  else begin
    data <= random_data;
  end
assume property ( output_data == data );
```

Formal picks a random value

Constrain output

How nondeterminism accelerates formal verification

- Randomness in formal
- Nondeterminism accelerates formal
- Wrap-up

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal

- Reducing the cone of influence
- Simultaneous testing
- Faster property development
- Handling inconclusives
- Abstractions

Design Symmetry

Code repetition
Parameterization may repeat code
Data Independence

```verilog
class arbiter {...
always @* begin
  case (pointer_reg)
  2'b00 : if (req[0]) grant = 4'b0001;
  else if (req[1]) grant = 4'b0100;
  else if (req[2]) grant = 4'b1000;
  else if (req[3]) grant = 4'b1000;
  else
    grant = 4'b0001;
endcase
end
```

Handling pointer_reg

```verilog
always @* begin
  case (req)
  2'b11 : if (req[3]) grant = 4'b1000;
  else             grant = 4'b0000;
endcase
```

Simultaneous testing

```verilog
always @* begin
  if (req[0]) grant = 4'b0001;
  else
    grant = 4'b0000;
end
```

Abstractions

```verilog
always @* begin
  if (req[0]) grant = 4'b0001;
end
```

Reducing the cone of influence

```verilog
always @* begin
  if (req[0]) grant = 4'b0000;
end
```

Adding Nondeterminism

```verilog
module arbiter {...}
always @* begin
  case (pointer_reg)
  2'b10 :
    grant = 4'b0100;
  else
    grant = 4'b0100;
endcase
```

 instructor pointer_reg

```verilog
module arbiter ...
always @* begin
  case (grant)
  2'b00 : snip_driver grant
  2'b01 : snip_driver pointer_reg
  2'b10 :
    any starting point okay
    netlist cutpoint pointer_reg
  else
    netlist cutpoint grant
  endcase
end
```

Free variable

```verilog
stopat pointer_reg
```}

Reduced Cone of Influence

```verilog
always @* begin
  case (grant)
  2'b00 : assume property ($onehot0(grant));
  else if (grant[0]) grant = 4'b1000;
  else if (grant[1]) grant = 4'b1000;
  else
    grant = 4'b0000;
endcase
```

Consider ...

```verilog
if (req[0]) grant = 4'b0001;
else if (req[1]) grant = 4'b0010;
else if (req[2]) grant = 4'b0010;
else if (req[3]) grant = 4'b0000;
```

This is equivalent to ...

```verilog
if (req[0]) grant[0] = 1;
else if (req[1]) grant[1] = 1;
else if (req[2]) grant[2] = 1;
else if (req[3]) grant[3] = 1;
```

Or ...

```verilog
req[pointer_reg] -> grant[pointer_reg]
```

Each request and grant is independent of each other

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal

- Reducing the cone of influence
- Simultaneous testing
- Faster property development
- Handling inconclusives
- Abstractions
Modeling with Free Variables

```verilog
module eccAssertions (input logic [7:0] data, input logic [12:0] ecc_from_dut, ...);

logic [12:0] expected_ecc;
assign expected_ecc[11:0] == {
  data[7],
data[6],
data[5],
data[4],
p8, 
data[3],
data[2],
data[1], p4, 
data[0], p2, p1 }
assign expected_ecc[12] = ^
  expected_ecc[11:0];
endmodule
```

Formal can test all values of data simultaneously!

Symbolic Constant

Symbolic constants do not change once initialized

```verilog
bit [7:0] index;
assume property ( ##1 $stable(index) );
assert property ( request[index] |=> grant[index] );
```

All indices proven at the same time
Simplifies modeling and reduces analysis effort

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal
- Reducing the cone of influence
- Simultaneous testing
- Faster property development
- Handling inconclusives
- Abstractions

Control Knobs

```verilog
// Used by formal tool to select good or bad packets
enum bit { FALSE = 0, TRUE = 1 } pkt_good;

pkt_good acts as a knob in the constraints
```
Setting Knobs

Control knob enables the formal target

```verilog
// Sequence to indicate the design is parsing the packet
property pkt_parsing;
!parse[*0:$] ##1 parse[*1:$] ##1 !parse;
endproperty
```

Simplifying Properties

```verilog
// Synthesizes N number of flops – adds extra state space
assert property ( req |-> ##N ack );
```

Start Signals

```
bit start;
enum bit [1:0] { IDLE, COUNT, DONE, ERROR } state;
```

How nondeterminism accelerates formal verification

- Reducing the cone of influence
- Simultaneous testing
- Faster property development
- Handling inconclusives
- Abstractions
Deep State Space Search

always @(posedge reset or posedge clock)
if (reset)
    count <= 0;
else if (count < 16'hffff)
    count <= count + 1;
else
    count <= 0;

always @(posedge clock)
begin:
    fail_eventually
    if (en1)
        array[address1] <= data_in; // Bounds check
    if (en2)
        array[address2] <= data_in; // Bounds check
    if (en3)
        array[address3] <= data_in; // Bounds check
end

Cut Points

always @(posedge reset or posedge clock)
if (reset)
    count <= 0;
else begin
    count <= count + 1;
end:

Free variable

Set proof depth = 5000
Fails at depth=512
Fails at depth=2048
Inconclusive at 5000

How nondeterminism accelerates formal verification

Nondeterminism accelerates formal
• Reducing the cone of influence
• Simultaneous testing
• Faster property development
• Handling inconclusives
• Abstractions

Counters

Counters have a large state space
A counter abstraction can reduce the state space for formal

always @(posedge reset or posedge clock)
if (reset)
    count <= '0;
else begin
    count <= count + 1;
end:

Correct initialization
Counter behavior (including wrapping)
Counter Abstraction

Nondeterminism inserted using free variables

```vhdl
always @ (posedge reset or posedge clock )
if ( reset )
  count <= 0;
else if ( count == MAX )
  count <= 0;
else
  count <= count + 1;
logic [3:0] increment;
assume property (count == $past(count) + increment) % MAX);
```

Free variables

```
netlist cutpoint count
snip_driver count
stopat count
Add cut point
```

Priority Arbiter Example

Requirements:

- j < k
- req[j] has higher priority than req[k]
- req[j] implies req[k] not granted until req[j] granted first

Simultaneous Checking

```
bit [${clog2(N)-1:0}] j, k;

// Priority encoded
assume property ( ##1 $stable(j) && $stable(k) );
// Priority encoded
assume property ( j < k );
// Keep within range
assume property ( k < N );

All combinations of j and k checked simultaneously!

assert property ( req[j] |-> !gnt[k] s_until_with gnt[j] );
```

How nondeterminism accelerates formal verification

- Randomness in formal
- Nondeterminism accelerates formal
- Wrap-up
Summary

Nondeterminism accelerates formal by …

- Fewer properties => faster property development
- Formal modeling => simultaneous testing
- Reducing state space => faster state exploration
- Abstraction => deeper state exploration
- Randomness => exhaustive testing

Thank you for attending

We hope you found this information helpful!
Track Session

VHDL Verification
Lonestar Ballroom – Salon D

We would be grateful if you could move to the track session as quickly as possible.
Notes
Jim Lewis
SynthWorks Design Inc
VHDL Verification Specialist

OSVVM in a Nutshell, VHDL’s #1 Verification Methodology

User Paper

Abstract
OSVVM is an advanced verification methodology that defines a VHDL verification framework, verification utility library, verification component library, scripting API, and co-simulation capability that simplifies your FPGA or ASIC verification project from start to finish. Using these libraries you can create a readable, powerful, and concise testbench that will boost productivity for either low level block tests (unit tests) or complex FPGA and ASIC tests.

OSVVM is developed by the same VHDL experts who have helped develop VHDL standards. We have used our expert VHDL skills to create advanced verification capabilities that provide:

- A structured transaction-based verification framework using verification components.
- A common, shared transaction API for address bus (AXI4, Avalon, …) and streaming (AXI Stream, UART) verification components.
- Improved readability and reviewability by the whole team including software and system engineers.
- Improved reuse and reduced project schedules.
- Buzz word features including Constrained Random, Functional Coverage, Scoreboards, FIFOs, Memory Models, error logging and reporting, and message filtering that are simple to use and work like built-in language features.
- A common scripting API to run all simulators. OSVVM scripting supports GHDL, NVC, Aldec Riviera-PRO and ActiveHDL, Siemens Questa and ModelSim, Synopsys VCS, and Cadence Xcelium.
• A Co-simulation capability that supports running software (C++) in a hardware simulation environment.
• Unmatched test reporting with HTML based test suite reports, test case reports, and logs that facilitate debug and test artifact collection.
• Support for continuous integration (CI/CD) with JUnit XML test suite reporting.
• A rival to the verification capabilities of SystemVerilog + UVM.

OSVVM has grown rapidly during the COVID years, giving us better capability, better test reporting (HTML and JUnit), and scripting that is simple to use (and works with most VHDL simulators). This presentation will show how these advances fit into the overall OSVVM Methodology.

Looking to improve your VHDL verification methodology? OSVVM provides a complete solution for VHDL ASIC or FPGA verification. There is no new language to learn. It is simple, powerful, and concise. Any VHDL engineer can write either tests or verification components.

Each piece/capability of OSVVM can be used separately. Hence, you can learn and adopt pieces as you need them.

Maybe your EDA vendor has suggested that you should be using SystemVerilog for verification. According to the 2022 Wilson Verification Survey [1], for both FPGA design and verification, VHDL is used more often than Verilog or SystemVerilog. Likewise, in the survey you will find that OSVVM is the #1 VHDL verification methodology.


Biography
Jim Lewis is an innovator and leader in the VHDL community. He has 30 plus years of design and teaching experience. He is the Chair of the IEEE 1076 VHDL Standards Working Group. He is a co-founder of the Open Source VHDL Verification Methodology (OSVVM) and the chief architect of the packages and methodology. He is an expert VHDL trainer for SynthWorks Design Inc. In his design practice, he has created designs for print servers, IMA E1/T1 networking, fighter jets, video phones, and space craft.

Whether teaching, developing OSVVM, consulting on VHDL design and verification projects, or working on the IEEE VHDL standard, Mr Lewis brings a deep understanding of VHDL to architect solutions that solve difficult problems in simple ways.
OSVVM in a Nutshell

OSVVM in a Nutshell

Copyright © 2020 - 2021 by SynthWorks Design Inc.

Distributing a pdf version of this document in whole or in part for individual usage is permitted.

Printing a copy of this document in whole or in part for individual usage is permitted.

All other rights reserved.

In particular, without express written permission of SynthWorks Design Inc.

You may not distribute printed versions of this document.

You may not alter, transform, or build upon this work.

You may not use any material from this guide in a classroom presentation,
tutorial, training, or classroom.

You must include this page in any printed copy of this document.

This material is derived from SynthWorks' class, VHDL Testbenches and Verification.

This material is updated from time to time and the latest copy of this is available at
http://www.SynthWorks.com/papers

Contact & Information

Jim Lewis, President
SynthWorks Design Inc.
11985 SK 128th Avenue
Tigard, Oregon 97223
503-390-4891
jim@SynthWorks.com

www.SynthWorks.com

OSVVM in a Nutshell

• Agenda
  • What is OSVVM?
  • OSVVM Verification Framework
  • Verification Components
  • Test Sequencer
  • Writing Directed Tests
  • Constrained Random Tests
  • Scoreboards
  • Functional Coverage
  • Intelligent Coverage Random
  • Protocol and Parameter Checks
  • Test Reporting
  • Scripts

Background

• About Jim Lewis
  • 30 years: VHDL Design and Verification
  • 20+ years: Active in IEEE VHDL WGs
  • 15+ years: REEL VHDL WG chair
  • Chief Architect of OSVVM
  • VHDL Consultant and Trainer for SynthWorks

• SynthWorks provides VHDL Training
  • Comprehensive VHDL Introduction
  • VHDL Coding for Synthesis
  • Advanced VHDL Testbenches and Verification – OSVVM Bootcamp

OSVVM is developed by the same VHDL experts who have helped develop VHDL standards.
**Why VHDL?**
- VHDL is #1 for FPGA Design and Verification
- From Wilson Research Group 2022 Functional Verification Survey
  - For FPGA design: 66% worldwide use VHDL
  - For FPGA verification: 56% worldwide use VHDL

![VHDL Survey Graph]

**Why OSVVM?**
- OSVVM is VHDL’s #1 Verification Methodology
- For FPGA Verification,
  - Worldwide: 28% use OSVVM = 50% of the VHDL FPGA users

![OSVVM Survey Graph]

**What is OSVVM?**
- Verification Framework
  - Transaction Interface & API
  - Verification Components
  - Test Sequencer (Test Cases)
- Verification Utility Library
  - Constrained Random, Scoreboards, Functional Coverage, Memory Models, Error tracking, and Message filtering, ...
- Verification Component Library
  - AXI4 Full, Lite, AXI Stream, UART, JTAG, DiffRam, ...
- Script Library
  - Tool Independent Scripts
  - One Script to Run them All
- Co-Simulation
  - Run Software in Hardware Simulator
  - Write tests in C++
- Test Reports
  - HTML, Test Suite, Test Case, Log Files
  - JUnit XML for CUCD tools

**OSVVM History**
- 1997: Transaction Framework, TblJHPkg
- 2006: RandomPkg, ResolutionPkg, ScoreboardPkg, MemoryPkg
- 2010: CoveragePkg
- 2011: RandomPkg, CoveragePkg
- 2015: AlertLogPkg, TranscriptPkg, MemoryPkg
- 2016: ScoreboardGenericPkg, TblJHPkg, ResolutionPkg
- 2016: AdvLite, AviStream, UART
- 2020: Scripting, Specification Tracking, MIF, Virtual Interfaces, AXI4 Full and AuxStream, both with Bursting
- 2021: Singleton Data Structures, HTML & JUnit XML reports
- 2022: HTML Log & Scoreboard Reports, Code Coverage Reports, Ethernet VC, Arrays of Transaction Interfaces
- 2023: Co-simulation of C++ Software in a Hardware Simulator, VC with delay randomization, Specification Tracking Part 2
OSVVM Verification Framework

- Looks identical to a SystemVerilog framework.
- Verification components (VC) implement interface signaling
- Test sequencer (TestCtrl) calls transactions - test case
- Each test case is a separate architecture of TestCtrl

**Framework Implementation**

```vhdl
library osvvm, osvvm_Axi4;
context osvvm.OSVVMContext;

entity TbaAxi4 is
end entity TbaAxi4;
architecture TestHarness of TbaAxi4 is
    signal ManagerRec : AddressBusRecType (Address (AXI_ADDR_WIDTH-1 downto 0),
    DataFromModel (AXI_DATA_WIDTH-1 downto 0),
    DataFromProc (AXI_DATA_WIDTH-1 downto 0));
begin
    osvvm.TbUtilPrk.CreateClock(Clk, period_Clk);
    osvvm.TbUtilPrk.CreateReset(Rreset, ...);
    DUT: DUT ( ... ) ;
    AxiManagerRec_1 : AxiManager (Rrecv, ...);
    DataRec_1 : DataRec (Rrecv, ...);
    DataRec_2 : DataRec (Rrecv, ...);
    TestCtrl_1 : TestCtrl (Rrecv, Rrecv, Rrecv, Rrecv);
    end TestHarness;
end architecture TestHarness;
```

3 Steps to VC Development

- Step 1: Transaction interface (ManagerRec, ...)
- Step 2: Transaction API (Write, Send, ...)
- Step 3: Verification components

**Step 1: Transaction Interface = Record**

```vhdl
type AddressBusRecType is record
  Addr : AddrType ;
  Ack : AckType ;
  Operation : AddressBusOperationType ;
  Address : std_logic_vector(max_o);
  AddrWidth : integer_max ;
  DataFromModel : std_logic_vector(max_o);
  DataFromProc : std_logic_vector(max_p);
  DataWidth : integer_max ;
end record AddressBusRecType ;
```

- The record is an "inout" port
- The "magic" is in the types and resolution functions (from ResolutionPkg)

Long term plan is to migrate to VHDL 2019 Interfaces
- Only requires a module view declaration for the record
Step 2: Transaction API = VHDL Procedure

```vhdl
procedure Read (
    signal TransRec : inout AddressBusType;
    iAdd : in std_logic_vector;
    var oData : out std_logic_vector;
    StatusMap : in boolean := FALSE
)
begin
    -- Put Transaction into Record
    TransRec.Operation <= READ_OP;
    TransRec.Address <= iAdd(Addr, TransRec.Address'length);
    TransRec.Address(Addr, iAdd'length);
    TransRec.DataLength <= iAdd length ;
    TransRec.StatusMap <= StatusMap ;

    -- Handshake with Verification Component
    RequestTransaction(iAdd, TransRec.Ack <= TransRec.Ack);
    -- Get Results
    oData <= safemap(transRec)
end procedure Read ;
```

Step 3: Verification Components

```vhdl
entity AxiManager is
    generic {
        tperiod_Clck : time := 10 ns;
        tpdt_Clck_Rready : time := 2 ns
    };
    port {
        clk : in std_logic;
        nReset : in std_logic;

        -- AXI Master Functional Interface
        AxBus : inout Axi4Rectype ;

        -- Testbench Transaction Interface
        TransRec : inout AddressBusType;
        Transaction Interface
    };
end entity AxiManager ;
```

Step 3: Verification Components

```vhdl
procedure ReadHandler : process
begin
    WaitForTransaction:
        if clk = Clk, Rdy = TransRec.Rdy, Ack = TransRec.Ack then
            -- Handle and execute the transaction
            case TransRec.Operation is
                when WRITE_OP =>
                    AxBus (TransRec.Address, TransRec.Data, -);
                when READ_OP =>
                    AxBus (TransRec.Address, TransRec.Data, -);
                when others =>
                    Other Transactions end case;
            end case;
        end if;
    end process TransactionHandler ;
```

Simplifying VC Development

- Observation: Some interfaces do the same transactions
- Address Bus Interfaces (AXI4, Avalon, ...) do read and write
- Streaming Interfaces (AXIStream, UART, ...) do send and get
- For these interfaces, Model Independent Transactions (MIT) define
  - Transaction Interface (record) and Transaction API (procedures)
  - Address Bus MIT (Basic subset)
    ```vhdl
    type AddressBusRectype is record . . . ;
    Write (AddrRec, iAddr, iData) ;
    Read (AddrRec, X'1111_1110', oData) ;
    ```
  - Stream MIT (basic subset)
    ```vhdl
    type StreamRectype is record . . . ;
    Send (TxRec, oData [], oParam) ;
    Get (RxRec, oData [], oParam) ;
    ```
Benefits of OSVVM MIT

- Accelerates VC Development, Test Writing, and Documentation
- Verification Component Developers
  - Re-use the transaction interface and API defined by MIT
  - Focus on writing VC behavior
- Test Writers
  - Similar VC use the same transaction API
  - Similar VC can share sequences of transactions
- Co-Simulation Interface
  - Supports all MIT based VC - including ones you write
- Documentation
  - VC only need to identify which transactions they support
- More Information is in user guides in OsvvmLibraries/Documentation/
  - Address_Bus_Model_independent_transactions_user_guide.pdf
  - Stream_Model_independent_transactions_user_guide.pdf

TestCtrl = Test Sequencer

```vhdl
entity TestCtrl is
begin
  xSec : out streamSecType;
  foSec : out streamSecType;
  rSec : out AddressBusSecType;
  rReset : in std_logic;
end TestCtrl;
```

architecture UartTx1 of TestCtrl is
begin
  ControlProc : process
  begin
    WaitForBarrier(TestDone, 5 ms);
    EndIfTestReports;
    std_now := end process;
    WaitForBarrier : process
    begin
      wait until rReset = '1';
      WaitForBarrier(TestInit);
    end process;
  end process;
end TestCtrl;

Test Initialization in ControlProc

```vhdl
ControlProc : process
begin
  SetTestName("UartTx1");
  TranscriptOpen;
  TranscriptMirror(TRUE);
  TRB <= NewID("TE");
  PdID <= NewID("UartRx_1");
  SB <= NewID("SB", ModelID);
  SetLogEnable(PASRED, TRUE);
  SetLogEnable(PdID, INFO, TRUE);
  WaitForBarrier(TestDone, 5 ms);
end process;
```

Aspects of a Test Sequencer

- Whole test in one file
- Control Process
  - Initialize & finalize test
  - One process per interface
  - Concurrent, just like design
- Tests
  - Calls to transactions
  - Easy to add and mix in
    - Directed Tests
    - Constrained Random
    - Scoreboards
    - Functional Coverage
  - Synchronization
  - Error Reporting & Messaging
OSVVM Makes Writing Tests Easy

- Call transactions such as Send, Get, and Check.
  
  ```
  TfProc : process
  begin
    Send (TxRec, X'10') ;
    Send (TxRec, X'11') ;
    WaitForBarrier(...) ;
  end process TfProc ;

  RxProc : process
  begin
    Get (RxRec, X'10') ;
    AffirmIfEqual (TID, RxRec) ;
    Check (RxRec, X'11') ;
    WaitForBarrier (TxDone) ;
  end process RxProc ;
  ```

- Test Output for AffirmIfEqual:
  ```
  ++ Log PASSED 10.00 Received: 10 at 2150 ns
  ++ Alert ERROR 10.00 Received: 08 /= Expected: 10 at 2150 ns
  ++ Benefit: improves readability. Simplifies writing self-checking tests
  ```

OSVVM Makes Randomization Easy

- Why Randomize?
  - Directed test of a FIFO (tracking words in FIFO):  

- Constrained Random test of a FIFO:  

- Key Benefits:
  - Generates realistic stimulus in a timely fashion (to write)
  - Ideal for large variety of similar items
  - Modes, sequences, network packets, processor instructions, ...

OSVVM Makes Randomization Easy

- Randomize a value in an inclusive range, 0 to 15, except 5 & 11
  ```
  Data1 := RV. RandInt ( Min => 0, Max => 15 ) ;
  Data2 := RV. RandInt ( 0, 15, Exclude => (5,11) ) ;
  ```

- Randomize a value within the set (1,2,3,5,7,11), except 5 & 11
  ```
  Data3 := RV. RandInt ( [1,2,3,5,7,11] ) ;
  Data4 := RV. RandInt ( 1,2,3,5,7,11, Exclude => (5,11) ) ;
  ```

- Weighted Randomization: Weight, Value = 0 .. N-1
  ```
  Data5 := RV. DistInt ( [1,2,3] ) ;
  ```

- Weighted Randomization: Value + Weight
  ```
  Data6 := RV. DistValInt ( [{1,71}, {2,5}, {3,11}] ) ;
  ```

By itself this is not constrained random.

OSVVM Constrained Random Tests

- Code Pattern + Randomization + Transaction Calls
  ```
  TfProc : process
  variable RV : RandomType ;
  while i <= 10000 loop
    case RV.DistInt ( [0,10,15] ) is
      when 0 => -- Nominal case 70%
        Nominal : 70%
        T := RV. RandSilv (0, 255, Data'length) ;
      when 1 => -- Parity Error 10%
        Stop Error 10%
        T := RV. RandSilv1 (255, Data'length) ;
      when 2 => -- Stop Error 10%
        Operation := UNAINT6_LOOP_ERROR ;
        T := RV. RandSilv1 (255, Data'length) ;
      when 3 => -- (3 and 4)
        Do Transaction
    end case ;
    Send (TxRec, TmD, Operation) ;
  end loop ;
  ```
Constrained Random and Checking?

For checking, RxProc could repeat the randomization from TxProc, however, this is tedious and potentially error prone.

TaProc : process
variable TX : ByteType;
variable RV : RandomType;
begin
for i in 1 to 10000 loop
    case RV.nextInt(1, 3) is
    end case;
    Send(TxRec, TX, Op);
    end loop;
    ... WaitForBarrier(TestDone);
end process TaProc;

RxProc : process
variable ExpO : ByteType;
variable RV : RandomType;
begin
for i in 1 to 10000 loop
    case RV.nextInt(1, 3) is
    end case;
    Check(TxRec, ExpO, Op);
    end loop;
    ... WaitForBarrier(TestDone);
end process RxProc;

Scoreboards

- Simplify self-checking when data is minimally transformed

OSVVM Generic Scoreboards

```java
package ScoreBoardGenericPkg is
  generic:
  type ExpectedType is ...
  type ActualType is ...
  function Match ( , ) return boolean;
  function expected_to_string ( , ) return string;
  function actual_to_string ( , ) return string;

  type ScoreBoardPType is ...
  procedure Send ( );
  procedure Push ( );
  procedure Check ( );
  procedure Pop ( );
  impure function Top ( ) return ExpectedType;
  impure function Empty ( ) return boolean;

  protected
    type ScoreBoardPT is protected ...
    end protected ScoreBoardPType;
  end ScoreBoardGenericPkg;
```

OSVVM Scoreboards: Generic Instance

```java
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
package ScoreBoardPkg_slv is new osvvm.ScoreBoardGenericPkg
  generic map ()
  (ExpectedType => std_logic_vector,
   ActualType => std_logic_vector,
   match => MetaMatch,
   expected_to_string => to_hexstring,
   actual_to_string => to_hexstring);
package ScoreBoardPkg_int is new osvvm.ScoreBoardGenericPkg
  generic map ()
  (ExpectedType => integer,
   ActualType => integer,
   match => equality,
   expected_to_string => to_string,
   actual_to_string => to_string);
```

Both in OSVVM library
Using OSVVM’s Scoreboard is Easy

OSVVM’s Data Structures (SB, FIFO, FC, and Alerts) use singletons
- Singletons use ordinary types and constructors (NewID)
- Easier than our older methodology, which uses protected types.

Functional Coverage

- What: Code that tracks that items in the test plan occur
- Tracks requirements, features, and boundary conditions

- Why?
  - With Randomization, how do you know what the test did?
  - Test Done = Functional Coverage and Code Coverage @ 100 %

- Item Coverage (aka Point Coverage)
  - Track relationships within a single object
  - Bin transfer sizes into: 1, 2, 3, 4-127, 128-252, 253, 254, 255

- Cross Coverage
  - Track relationships between independent objects
  - Has each set of registers been used with each input of an ALU?
  - Why not just use code coverage?
  - Code coverage tracks code execution
  - Misses anything not in code (bins, uncorrelated items)

CoveragePkg

- CoveragePkg simplifies coverage definition, collection, and reporting
- Internally it has a data structure and configuration parameters
- Implemented as a singleton in CoveragePkg
- The singleton API defines the coverage capabilities

```haskell
function GenBin ( ... ) return CovBinType :
  type CoverageIDType is ... .
  impure function NewID(Name: string; ... ) return CoverageIDType ;
  procedure AddBins ( ID : CoverageIDType; CovBin : CovBinType ) ;
  procedure AddCross ( ID : CoverageIDType; R1, R2, ... , CovBinType ) ;
  procedure ICover ( ID : CoverageIDType; val : integer ) ;
  procedure ICover ( ID : CoverageIDType; val : integer, vector ) ;
  impure function IsCovered ( ID : CoverageIDType ) return boolean ;
  procedure WriteBin ( ID : CoverageIDType ) ;
  procedure WriteCovHoles ( ID : CoverageIDType ) ;
  ... 
```

OSVVM Functional Coverage is Easy

- For the UART, we track the following items

<table>
<thead>
<tr>
<th>Condition</th>
<th>Break Error</th>
<th>Stop Error</th>
<th>Parity Error</th>
<th>Done Error</th>
<th>Integer Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal Transfer</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Parity Error</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Stop Error</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Parity &amp; Stop Error</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Break Error</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>9.15</td>
<td></td>
</tr>
</tbody>
</table>
OSVVM Functional Coverage is Easy

```
architecture CR1 of TestCtrl is
signal RxConv : Coverage(type);
end architecture;

RxProc : process
begin
RxConv <= NewID("RxConv", TB_ID);
wait for 0 ns;
AddBin(RxConv, GenBin(1)) ; -- Normal
AddBin(RxConv, GenBin(3)) ; -- Parity Error
AddBin(RxConv, GenBin(5)) ; -- Stop Error
AddBin(RxConv, GenBin(7)) ; -- Parity + Stop
AddBin(RxConv, GenBin(15, 1)) ; -- Break

for i in 1 to 10000 loop
Get(RxRec, RxO, RxOp);
Check(SIG, Bad, RxOp);
Cover(RxConv, to_integer(RxOp));
end loop ;
end ...
```

Functional Coverage with OSVVM is as simple and concise as language syntax.

OSVVM Intelligent Coverage Randomization

```
TsProc : process
constant StimConv : CoverageIDType :=
end process;

StimConv := NewID("StimConv", TB_ID);
wait for 0 ns ;
AddBin(StimConv, "NORMAL", 7000, GenBin(1)) ; -- Coverage goals =
AddBin(StimConv, "FAIRY", 1000, GenBin(3)) ; -- Randomization
AddBin(StimConv, "STOP", 1000, GenBin(5)) ; -- Weights

for i in 1 to 10000 loop
iOperation := GetRandPoint(StimConv) ;
if iOperation is when 1 => . . . -- Nominal 70%
when 3 => . . . -- Parity 10%
. . .
end case :
Push(SIG, Data, Operation) ;
Send(TxRec, Data, Operation) ;
Cover(StimConv, iOperation) ;
wait for Idle * DAST_BAND_PERIOD_115200 ;
end loop ;
```

Intelligent Coverage goes beyond what SV does.

OSVVM Protocol and Parameter Checkers

```
SimultaneousAccessCheck : process
begin
wait on ICE, IWE, IHE :
AlertIf((iceAlertID = ice and iwe and ihe) = '1',
   "ICE, IWE, and IHE are all active") ;
end process SimultaneousAccessCheck ;
```

- Alerts signal errors and keep counts in the AlertLog data structure
- Alert Levels: FAILURE, ERROR (default), WARNING
- Controls: StopCount, PrintCount, Enable/Disable
  - SetAlertStopCount(ERROR, 20) ; -- Stop when 20
  - SetAlertPrintCount(CpuID, ERROR, 10) ; -- Limit printing
  - SetAlertEnable(WARNING, FALSE) ; -- Disable Alerts
  - Alerts are enabled by default and rarely disabled

OSVVM Logs Simplify Debug

```
Logs are conditional printing

TmProc : process
begin
Log(TbID, "Sequence 1 Starting", ALWAYS) ;
Log(TbID, "Test Last Failed Here", DEBUG) ; -- Disabled
Log(TbID, "Test Log Disabled Here", DEBUG) ; -- Disabled
Log(TbID, "Sequence 1 Starting at 2200 ns
```

- Log output for above
  - Log ALWAYS to TB, Sequence 1 Starting at 2200 ns
- Message with level DEBUG does not print since it is disabled
Test Finalization

```c
ControlProc : process
begin
  SetTestName("DartRxl");
  . . . -- Test Initialization
  WaitForBarrier(TestDone, 5 ms);
  AlertIf(TBID, NOW >= 5 ms, "Test timed out");
  AlertIf(TBID, not Empty(SB), "Scoreboard not empty");
  AlertIf(TBID, GetAffirmCount X 1, "Checked < 1 items");
  AffirmDiff("DartRxl.log", "Checked/DartRxl.log");
endofTestsReports;
std.env.stop;
wait;
end process ControlProc;
```

OSvvm Test Watch Dog

- Purpose: Stop a process until all processes have reached the barrier
- signal TestDone : integer barrier := 1;

```c
ControlProc : process
begin
  SetTestName("DartRxl");
  . . .
  WaitForBarrier(TestDone, 5 ms);
  . . .
  ReportAlerts;
  std.env.stop(GetAlertCount);
end process ControlProc;
```

Including the OSVVM Library is Easy

- OSVM includes numerous packages.
- To simplify this, OSVVM library provides context declarations

```c
library osvm;
  context osvm.OsvmContext; -- All OSVM packages
  context osvm_axi4.Axi4Context; -- AXI4 Full VC
  context osvm_axi4.Axi4LiteContext; -- AXI4 Lite VC
  context osvm_axi4.AxiStreamContext; -- AXI Stream VC
  context osvm_uart.UartContext; -- UART VC
  context osvm_dpram.DpramContext; -- Dpram VC
```

My Scripts Before OSVVM

```c
set DER_SRC [file dirname [status file]]
set LIB_NAME osvm_uart
if { ![file exists ./VHDL_LIBS/$LIB_NAME.vhd] } {
  vcd -2008 -work $LIB_NAME $DER_SRC/TestCtrl.s.vcd
  vcd -2008 -work $LIB_NAME $DER_SRC/TestCtrl.s.tcl
  vcd -2008 -work $LIB_NAME $DER_SRC/TestCtrl.s.dsh
  vcd -2008 -work $LIB_NAME $DER_SRC/TestCtrl.s.dph
  add wave -r ./VHDL_LIBS/$LIB_NAME/tcl
  run 40 ms
}
```

- Issues
  - Need help (TCL code) to find the source directory
  - Simulator Specific
  - Simulator API repeats the same information on many calls
Why is EDA Scripting Hard?

- Some blame TCL
- Issues
  - Simulator needs to run in a specific directory
  - Settings and Library information are in a *.ini or *.cfg
    - If not, the library info must be respresented on tool start
    - Hence, if you use "cd", you lose this information
  - Scripts need to be co-located with verification IP
    - Hence, they need directory information
  - The simulator API fundamentally misunderstands the VHDL work library
    - Work is not a name for a library
    - Work is the shorthand for the current library

OSVVM Scripting

- Procedure based API that runs on top of TCL

```tcl
library osvvm_TbsCart
analyze ./testbench/TestCtsl_e.vhd
analyze ./testbench/TbsStart.vhd
analyze ./testbench/TbsStart_SendSetl.vhd
simulate TbsCart_SendSet1
```

- Benefits
  - Simple, just like a list of source files...
  - Except we also get the powers of TCL
  - Settings (like current working library) are remembered
  - Paths are relative to script location to facilitate moving pieces of projects
  - One Simulator Script API for
    - GHDL, NVC, Aldec, Siemens, Synopsys VCS, Cadence Xcelium

Calling OSVVM Scripts

- `build`: Call a .pro file from command line or CI

```bash
build .../OsvvmLibraries/OsvvmLibraries.pro
build .../OsvvmLibraries/RunDemoTestsWithCoverage.pro
```

- `build + EndOfTestReports` generates the HTML and Junit reports
- `include`: Call a *.pro file from another *.pro file

```tcl
# Ax14.pro
include ./common/common.pro
include ./Ax14Lite/Ax14Lite.pro
include ./Ax15Stream/Ax15Stream.pro
include ./Ax14/Ax14.pro
```

- Use Build and Include rather than TCL's source or EDA vendor's do
  - Used to make paths relative to directory from which the scripts run

OSVVM Test Completion Message

- `EndOfTestReports`

```
$ DONE PASSED Test_EarthRx_1 Passed: 48 Affirmations Checked: 68 at 100100100 ns
$ DONE FAILED Test_EarthRx_1 Total Error(s) = 7 Failures: 0 Errors: 7
Warnings: 0 Passed: 41 Affirmations Checked: 48 at 100100100 ns
```

- Default
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `OSVVM`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `Vu`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `GAX8b`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `AxiMaster_I`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `AxiMaster_I Data Bus`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `AxiMaster_I Protocol`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `AxiSlave_I`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `AxiSlave_I Data Bus`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `AxiSlave_I Protocol`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `UART_1`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `UART_2`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `UART_3`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `UART_4`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
- `UART_5`
  - Failures: 0 Errors: 0 Warnings: 0 Passed: 0
Build Summary Mini-Report

- When a build finishes, a single line mini-report is produced.

Build Error: Sim_Demo PAIRED, Passed 19, Failed 3.
Skipped: 6, Analyse Errors: 0, Simulate Errors: 0, Build Error Code: 0

- If there are errors, this is the first place we see an indication.

Build Summary Report

Created by Scripts = EndOfTestReport

- Build Status
  - Summarizes all tests run
  - Link to Simulation Transcript
  - Both list and HTML
  - Link to code coverage

- Test Suite Summaries
  - Test Suite = multiple test cases
  - Summarizes Pass/Fail+

- Test Case Summaries
  - Test Case = Testbench
  - Identify failing tests
  - Links to Test Case Reports

Build Summary Report with Errors

Test Case Report

- Links
  - Alert Report
  - Functional Coverage Report
  - Scoreboard Report
  - Simulation Results
  - Test Case Trans/Wl
  - Link to Build Summary

- Alert Report
  - Setup/Results
  - Results (Results)

- Functional Coverage Report
  - Report for each IC model in testbench (each instance)
A separate table is created for each scoreboard instance
Each row in the table has statistics for a single scoreboard
Tables for Scoreboard_slv and Scoreboard_int are automatically generated
Use WriteScoreboardHaml to generate reports for user created scoreboards
VHDL Transcript File

Getting OSVVM & Running Scripts
- Get the sources:
  git clone --recursive https://github.com/OSVVM/OSVMLibraries
- Alternately, a zip file is at: osvvm.org/downloads
- Initialize the simulator - see Documentation/Scripts_user_guide.pdf
  file path sim ; # In directory containing OSVMLibraries
cd sim
  source ../OSVMLibraries/Scripts/StartUp.tcl
- Build all OSVVM and Run All VC Tests
  build $OSVMLibraries/OSVMLibraries.pro
  build $OSVMLibraries/RunAllTests.pro
- Each VC has a RunAllTests and RunDemoTests

All you need is ... OSVVM
- Benefits
  • Powerful and Concise – rivals other verification languages
  • Unmatched reuse throughout the entire verification process
  • Unmatched report capability with HTML for humans and JUnit XML for CI
  • Tests are Readable and Reviewable by All
  • Adapt incrementally as needed
  • Tests and VC can be written by any VHDL Engineer

SynthWorks VHDL Classes
Comprehensive VHDL Introduction 4 Days - beginners class
http://www.synthworks.com/comprehensive_vhdl_introduction.htm
A design and verification engineer’s introduction to VHDL syntax, RTL coding, and testbenches. Students get VHDL hardware experience with our FPGA based lab board.

Advanced VHDL Testbenches and Verification - OSVVM Boot Camp - 5 days
http://www.synthworks.com/vhdl_testbench_verification.htm
Learn the latest VHDL verification techniques including transaction based modeling, self checking, scoreboard, memory modeling, functional coverage, directed, algorithmic, constrained random, and intelligent testbench test generation. Create a VHDL testbench environment that is competitive with other verification languages, such as SystemVerilog or SystemC. Out techniques work on VHDL simulators without additional licenses and are accessible to RTL engineers.

VHDL Coding for Synthesis 4 Days
http://www.synthworks.com/vhdl rtl synthesis.htm
Learn VHDL RTL (FPGA and ASIC) coding styles, methodologies, design techniques, problem solving techniques, and advanced language constructs to produce better, faster, and smaller logic.
OSVVM Resources

- Documentation
  - HTML: https://osvvm.github.io/Overview/osvvm1About.html
  - PDF: OsvvmLibraries/Documentation - in OSVVM release
- Forum: https://osvvm.org

Recorded Webinars
- OSVVM: Leading Edge Verification for the VHDL Community
  - https://www.youtube.com/watch?v=KvMglbJPHNI
- Faster than Lite Verification Component Development with OSVVM
- OSVVM’s Test Reports and Simulator Independent Scripting
- Advances in OSVVM’s Verification Data Structures

- Jump start your VHDL verification effort with training
  - Advanced VHDL Testbenches and Verification – OSVVM Boot Camp
  - https://synthworks.com/vhdl.testbench.verification.htm

Why OSVVM?

- OSVVM is VHDL’s #1 Verification Methodology
- For FPGA Verification,
  - Worldwide: 28% use OSVVM = 50% of the VHDL FPGA users

FPGA Verification Methodology per Wilson Survey

Tessolve is the leading engineering service/solution provider with 3000+ employees worldwide and a full breadth of pre-and post-silicon expertise. Tessolve provides a one-stop-shop solution with full-fledged hardware and software capabilities, including its advanced silicon and system testing labs. We have Test labs in India, the US, Malaysia, and Singapore.

Tessolve offers complete Turnkey ASIC Solutions, from design to packaged parts. We are actively investing in the R&D center of excellence initiatives such as 5G, mmWave, high power PMICs, HSIO, HBM/3D/Chiplets, system-level tests, advanced verification methodologies, and others.

Tessolve also offers end-to-end product design services in the embedded domain from concept to manufacturing under an ODM model with application expertise in Avionics, Automotive, Data Centre/Enterprise, Industrial/IoT, and Wireless segments. We are ISO 9001:2015 certified and our Embedded team is ISO9001 & EN9100 Quality certified.

Visit www.tessolve.com for more information.
Notes
We hope you have found the conference interesting and informative.

Slides and recordings will be available on the Tessolve Website.

https://www.tessolve.com/verification-futures/vf2022/