Welcome to the 10th Anniversary edition of Verification Horizons! It’s hard to believe we’ve been doing this long enough to reach such a significant milestone, but it’s been a great ride. For me personally, this issue marks two significant milestones as well.

My son and I reached the pinnacle of our Boy Scout experience this past summer when we successfully completed a 10-day 80-mile hike at the Philmont Scout Ranch in New Mexico, including reaching the 12,441 ft. summit of Baldy Mountain. Please indulge me in a bit of fatherly pride when I tell you that the highlight of the trip was sharing the experience with my son. The other major milestone is that my daughter has just started high school. She’s quickly gotten involved in the Drama Club and was cast in the fall play, which is a milestone I’m sure she’ll remember for a long time. We are very proud of how well she’s handled this transition in her life, despite the fact that it makes my wife and me feel old that our youngest is now in high school.

As with everything, milestones mark the end of one phase and the beginning of another. As much as we’ve enjoyed sharing so many articles about verification techniques and technology with you over the years, we look forward to continuing that mission as we advance to our next milestone.

So, we begin this issue with “Memories are Made Like This” by my colleague Mark Peryer of our Verification IP group. This article explains how the new Model Generator utility in our Questa® VIP (QVIP) library can be used to generate DDR and LPDDR models that match the behavior of over 750 commercially available memory devices. The easy-to-use graphical utility lets you specify the device you want and creates a wrapper module around the Mentor DDR QVIP BFM to provide an accurate and sophisticated simulation model to support your verification effort.
In “A New Stimulus Model for CPU Instruction Sets,” my colleagues Staffan Berg and Mike Andrews show how to create a portable stimulus model to verify that a specific implementation of a processor is fully compliant with the specification. Using a graph lets you describe the entire instruction set more concisely and efficiently than relying on SystemVerilog constraints, and by being at a higher level of abstraction also allows multiple implementations to be derived from a single description. Some of the work being done in the Accellera Portable Stimulus Working Group is building on these concepts.

We next take a look at post-silicon debug in “On-Chip Debug – Reducing Overall ASIC Development Schedule Risk” by my colleague Eric Rentschler. If you’ve ever wondered how to see what’s happening inside an FPGA or ASIC and how to debug the kinds of “long-latency” bugs that may slip through simulation, this article is for you. You’ll learn how our Certus™ debug solution automates the creation of debug infrastructure in your design and provides visibility and analysis capability that feels just like your simulation-based debug environment.

Our friend Lauro Rizzatti next brings us up-to-date in the finale of his three-part series “Hardware Emulation: Three Decades of Evolution.” In this installment, we see how emulation grew to where it is now used to debug hardware and develop software in parallel. This exhaustive case study will hopefully provide you with a solid foundation to help you decide how to incorporate emulation into your next verification project, if you haven’t already.

In our Partners’ Corner this issue, we begin with our friends at Agnisys who explain how “QVIP Provides Thoroughness in Verification.” This article is a great case study of the benefits in using Questa VIP to reliably verify many different configurations of AMBA® slave modules generated by their iDesignSpec™ tool. Most importantly, it allowed them to prove to their customers that their generated design IP is compliant with the standard.

Next, our friends at Oski Technology present “Minimizing Constraints to Debug Vacuous Proofs,” which explains how to improve formal verification productivity by isolating a minimal set of constraints necessary to identify failures and eliminate false positive results. If you’re using formal verification and running into this problem, you should definitely take a look.

We conclude this issue with our friends at SyoSil Aps discussing “A Generic UVM Scoreboard.” Those of you who use UVM are no doubt aware that the basic scoreboard component in the standard library is of limited usefulness. For cases where you need a more robust and scalable scoreboard, this article will explain how theirs works, and how you can download and use it on your next project.

I’d like to take this opportunity to thank Rebecca Granquist, who is really the “wizard behind the curtain” for Verification Horizons. She has made my job as Editor so much easier over the years and is a huge part of whatever success we’ve achieved. Looking back on the last ten years, I can honestly say that, professionally, it really doesn’t seem that long. When you have a job that you enjoy, as they say, time really does fly.

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons
Table of Contents November 2015 Issue

Page 4
Memories Are Made Like This
by Mark Peryer, Verification IP Architect, Mentor Graphics

Page 8
A New Stimulus Model for CPU Instruction Sets
by Staffan Berg, European Applications Engineer, and Mike Andrews, Verification Technologist, Mentor Graphics

Page 12
On-Chip Debug—Reducing Overall ASIC Development Schedule Risk
by Eric Rentschler, Chief Validation Scientist, Mentor Graphics

Page 15
Hardware Emulation: Three Decades of Evolution—Part III
by Dr. Lauro Rizzatti, Verification Consultant, Rizzatti LLC

Partners’ Corner

Page 19
QVIP Provides Thoroughness in Verification
by Kiran Sharma, Sr. Verification Engineer and Vipin Kumar, Sr. Verification Engineer, Agnisys Technology Pvt. Ltd.

Page 21
Minimizing Constraints to Debug Vacuous Proofs
by Anshul Jain, Verification Engineer, Oski Technology

Page 26
A Generic UVM Scoreboard
by Jacob Andersen, CTO, Kevin Seffensen, Consultant and UVM Specialist, Peter Jensen, Managing Director, SyoSil ApS
INTRODUCTION
One of the most common requirements for the verification of a chip, board or system is to be able to model the behaviour of memory components, and this is why memory models are one of the most prevalent types of Verification IP (VIP).

Memory models have two main functions. The first is to store information in a data structure so that it can be written, retrieved and updated. The second is to provide a signal level interface which allows access to the storage array using a pre-defined protocol (see figure 1 for a representative functional block diagram). For effective verification the model should also check that the signal level protocol from the host interface is behaving correctly, provide a backdoor access to the storage array and provide a means of collecting functional coverage.

Over the years, memory devices such as Dynamic RAMs (DRAM) have driven semiconductor process development as manufacturers have sought to create faster memories with a greater storage capacity through the use of smaller feature sizes. Over time, several different memory protocols have been developed to optimise the various trade-offs between device access speed and packaging costs, and today, one of the most widely used protocols is the Double Data Rate (DDR) memory protocol, and its low power variant, LPDDR. The DDR protocol is designed for high speed access, with data being transferred on both edges of the clock. However, in practice, this high level of throughput can only be achieved by managing the DDR accesses so that the time and power expensive operations such as activate and pre-charge can be taking place in one part of the memory whilst a data transfer takes place with another part of the memory. It is the task of a DDR controller to order the operations taking place in the DDR device to maximise throughput, and in order to verify the complex trade-offs a sophisticated memory model is needed.

In order to address the need for accurate DDR models, the Model Generator utility was introduced in the 10.4b release of Mentor’s Questa VIP (QVIP) library. The function of the Model Generator is to generate DDR and LPDDR models that match the behaviour of commercially available memory parts.

MODEL GENERATOR
QVIP Model Generator is an easy to use graphical utility that allows a user to specify and create a DDR or LPDDR model that will match the behaviour of over 750 commercially available devices. Underlying Model Generator is a code generator that supports all types of DDR and LPDDR models as defined in the JEDEC JESD79 and JESD209 specifications, with all possible configurations of device width, density and speed. The models are generated based on a memory manufacturer’s part number that represents an entry in a database that contains the details of each target memory device. The generated model is implemented as a Verilog module that is wrapped around the Mentor DDR QVIP.

To create a part in Model Generator GUI, the user first has to choose the DDR type and the manufacturer, this narrows their search down to a list of available devices. Then the user selects the required device from those part numbers available in the reduced list and then they press the generate button to write out the code for the device model. Figure 2, shows a screen shot of the part number selection pane for Micron DDR3 memories, the table entry row in blue is the one that has been selected. Table 1 lists the memory devices available from the various memory vendors that are supported by Model Generator.

The Model Generator software allows the user to create several models in a session, and also to specify the name of the generated model.

GENERATED MEMORY MODELS
The DDR/LPDDR models generated by Model Generator are intended to be instantiated in a VHDL, Verilog or
SystemVerilog/UVM testbench module as a component module. The models are reactive, in other words, they respond to the signalling on the protocol wires in order to determine how to behave and do not require any user stimulus in order to provide their functionality.

The models have a port map that corresponds to the memory device selected, and in some cases the models have several DDR/LPDDR interfaces since the selected memory device package contains several die. Each type of DDR and LPDDR has a different pin interface, and depending on the protocol there are several data bus width options per device, some of which have a further influence over the pin out. The generator has built-in rules that ensure that the right pin out is used. Most of the signal pins are simple inputs or outputs, but the data pins and their strobes are bidirectional, and the model switches between driving or receiving on these pins according to the current mode of operation.

![Figure 3. Model Generator memory model functional diagram](image)

<table>
<thead>
<tr>
<th>DDR Manufacturer</th>
<th>DDR2</th>
<th>DDR3</th>
<th>DDR4</th>
<th>LPDDR2</th>
<th>LPDDR3</th>
<th>LPDDR4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Micron</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
</tr>
<tr>
<td>Samsung</td>
<td>NA</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>NA</td>
</tr>
<tr>
<td>SK-Hynix</td>
<td>NA</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>NA</td>
</tr>
<tr>
<td>ESMT</td>
<td>Available</td>
<td>Available</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>ETRON</td>
<td>Available</td>
<td>Available</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>

Table 1. Available memory device types by manufacturer

Figure 2. Screen shot of the Model Generator GUI
The models are wrapped around the DDR QVIP BFM – see figure 3 for a functional block diagram of the features of the Model Generator memory model. The DDR QVIP BFM is configured by code inside the module to provide the correct functionality for the part selected in the generator. In addition to modelling the behaviour of the DDR memory, the QVIP BFM also has built-in assertions for protocol violations and will report any errors that occur.

**Transactional Features**

The generated models support two additional features that are normally only available in the context of UVM testbenches – functional coverage and transaction reporting. The functional coverage monitor is enabled by setting the models `ENABLE_FUNC_COV` parameter to a ‘1’. The functional coverage is collected by taking advantage of the transaction recognition features of the DDR QVIP BFM. Enabling the functional coverage monitor gives users of VHDL and Verilog testbenches the opportunity to take advantage of the built-in functional coverage to check which aspects of DDR or LPDDR functionality they have tested with their stimulus. The transaction reporting function is enabled by setting the models `ENABLE_TXN_LISTENER` parameter to a ‘1’. The transaction reporter prints a record to the transcript every time a transfer takes place over the DDR interface. Both of these features need to be enabled by the user, since they are disabled by default to optimise simulation performance.

The default behaviour of the models is that they are configured to work in the absence of a UVM testbench. If the `NO_UVM` parameter is set to ‘0’, then the functional coverage monitor and the transaction logger participate in the UVM phases. There is no support for creating a DDR memory agent in a UVM env since the DDR BFM needs no active stimulus.

Another transactional feature built into the models is the ability to use the hierarchical transaction viewing feature of the DDR QVIP to debug DDR activity via the Questa waveform view – see figure 4 for an example. This feature takes advantage of the built-in transactional objects that are available across the QVIP library.

**Backdoor API**

The models also support a backdoor API which enables the memory data storage array to be accessed directly without using the DDR signal interface. The backdoor API is intended to allow the user to either pre-load areas of memory with data, or to examine stored data content. The available API calls are summarised in table 2, and can be called using a hierarchical reference to the memory model.

There are several applications for this back door access functionality. If the design under test (DUT) contains one or more processors, then the DDR would provide the storage...
for the software binaries and the backdoor write API would be used to transfer a copy into the memory model. If the design contains a DMA function, then an area of memory would be initialised through the back door write API with a block of data containing a signature that would be copied to other places in memory during the course of the test case. Another possibility could be that the DUT has a data processing function, in which case the DDR memory would be initialised with the starting data on which the design would perform a transform and then write the result to another area of memory. The backdoor write APIs take an address and an array of bytes as their arguments. The data array content is written to consecutive locations starting at the given address.

Working in the opposite direction, the backdoor read APIs allow the content of the memory to be read back which can be useful for checking whether a data processing application has worked correctly, or that a known sequence of data transfers has completed successfully. The backdoor read APIs use a length argument to determine how many bytes are read back, starting from the location addressed by the address argument.

Finally, the dump_memory_content() API allows the DDR model to report on all locations accessed during the course of the simulation, listing bank, row and column addresses together with data content.

**CONCLUSION**

The Model Generator utility, released as part of the QVIP 10.4b library, supports models for more than 750 DDR and LPDDR commercial memory devices. The generated models provide comprehensive verification support for DDR and LPDDR memory interfaces, with a pin level interface, a complete set of protocol checking assertions, a functional coverage monitor and transaction reporting.

Model Generator proves an easy (EZ) way to generate an accurate and sophisticated simulation model for a standard part that can quickly be integrated into any Verilog or VHDL testbench.

### Backdoor API Call Summary

<table>
<thead>
<tr>
<th>Backdoor API Call Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>backdoor_wr_by_address(address, data[]);</td>
</tr>
<tr>
<td>backdoor_wr_by_brc(bank, row, column, data[]);</td>
</tr>
<tr>
<td>backdoor_rd_by_address(address, length, data[]);</td>
</tr>
<tr>
<td>backdoor_rd_by_brc(bank, row, column, length, data[]);</td>
</tr>
<tr>
<td>dump_memory_content(mem_data_content[]);</td>
</tr>
</tbody>
</table>

*Table 2. Model Generator backdoor API summary*
INTRODUCTION
Verifying that a specific implementation of a processor is fully compliant with the specification is a difficult task. Due to the very large total stimuli space it is difficult, if not impossible, to ensure that every architectural and micro-architectural feature has been exercised. Typical approaches involve collecting large test-suites of real SW, as well as using program generators based on constrained-random generation of instruction streams, but there are drawbacks to each.

SystemVerilog Constrained-Random (CR) ‘single class with constraints on all possible fields approach’ can be unwieldy and hard for solvers to handle, especially when dealing with a series of instructions with many for each constraint. Slow solving performance, testbench bugs hidden in the complexity, or simply not having the flexibility to fully describe the wanted scenarios for verification are quite common issues. When the problem is adequately modeled, achieving significant coverage is close to impossible.

There are custom options other than SV CR – creating an instruction stream generator from scratch, for example C++ may seem like a viable option, but the complexity and vast solution space quickly makes this approach unmanageable. Moreover, languages like C++ have no notion of functional coverage, which makes it hard or impossible to assess when the verification is complete.

In this article we present an alternative approach, using a portable, environment independent model of the stimulus. The CPU instruction set, and various contexts in which the instructions appear, are all modeled by a graph.

LEVERAGING A NEW PORTABLE STIMULUS APPROACH
Graph or rule based stimulus descriptions have been used for a long time in software and recently in hardware, including in CPU verification for modeling instruction streams in RT-level verification.

The key elements of this methodology are as follows:
- A declarative graph is used to completely model the Instruction set
  - Compact and intuitive format – well known in the SW community
  - Based on Accellera Proposed Standard for Portable Stimulus
- Separate Coverage model describes scenarios and corner-cases
  - E.g. Combinations of opcode, source and destination, interesting sequences of instructions...
- Graph is loaded in simulation together with DUT and traversed systematically to fulfill the coverage goals

The graph is described in a compact and intuitive format (BNF-like) – well known in the SW community. The format is based on Accellera Proposed Standard for Portable Stimulus – meaning that stimulus is independent of the simulation environment so the same model can theoretically be used with DUTs at any level in any language (or even used for other code generation or documentation purposes).

Figure 1. Sparc V8 Instruction set graph (Top level)
MODEL EXAMPLE
Here’s an example of modelling one of the instruction types (SETHI/BRANCH) using SystemVerilog, compared with the same example using the graph-based notation. As we can see, even for this simple example the result is a set of nested if-statements and complex constraints, which quickly makes it hard to debug and impacts solver performance in a negative way.

As a comparison, a single rule in the graph-based notation completely describes all forms this particular instruction type can take:

```
constraint format2_type_c {
  if (op == format2_op) {
    if (op2 == sethi_op2 && op2_subtype == 0) {
      i == 0; cond == 4'h0; imm22 == 22'h0;};
  }
```

Figure 2. Constraint for Sparc V8 SETHI instruction (Partial)

The visualization of this particular rule looks like the graphic in figure 4:

By building a hierarchy of these simple rules it is easy to create a highly structured model of any instruction set in a very compact and readable format.

SIMULATION INTEGRATION
One key element of the graph notation is the interface. An interface is an abstraction of the connection between the graph and the rest of the verification environment. By separating the connection from the graph implementation it is possible to achieve a high degree of horizontal re-use, since the same graph can drive many different verification environments. In a UVM environment for example, a graph can be used to fill the member fields of a UVM sequence item, replacing the calls to randomize() as shown below. The same graph could also be used to call functions in a library of C/C++ functions running on an embedded CPU.

```
// m_curr_instr.randomize();
m_instr_graph.get_inst(m_curr_instr);
```

Figure 5: Integration into a SystemVerilog C-R testbench
BENEFITS FOR FUNCTIONAL COVERAGE
The graph-based description can be augmented with information about the specific verification goals, as described perhaps by functional coverage metrics in SystemVerilog. Smart traversal algorithms can ensure that all instruction types are broadly exercised, hitting all corner cases efficiently.

The table below was taken from trial runs using a subset of the Sparc V8 instruction set that compared the two approaches.

In each case # of instructions’ is both the number of coverage bins and the number of instructions actually generated during the simulation.

<table>
<thead>
<tr>
<th>TESTCASE</th>
<th># OF INSTRUCTIONS</th>
<th>GRAPH COVERAGE</th>
<th>C-R COVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>All Opcodes</td>
<td>74</td>
<td>100%</td>
<td>61%</td>
</tr>
<tr>
<td>Arith Op X Dest reg</td>
<td>1600</td>
<td>100%</td>
<td>54%</td>
</tr>
</tbody>
</table>

Table 1. Graph based simulation coverage vs Constrained-random

MODELING PIPELINE CONSTRAINTS
Many modern GPU’s and DSP’s are based on Very Long Instruction Word (VLIW) architectures, where one instruction word can contain multiple parallel instructions, all executed in the same cycle(s). Due to the particular pipeline implementation and resources, there might be micro-architectural constraints that apply. For example, it might be possible to combine an ADD and a MULT instruction, but not two MULT since the pipeline only has one multiplier.

To model pipelines or sets of parallel instructions, a slightly different graph can be used, where each instruction is stored and the next one is generated following rules defined.

A new level of hierarchy is created as described earlier with two instances of a sub-graph that includes valid instructions that can be executed in parallel with another.

BRANCHES AND LOOPS
One of the difficulties using C-R to generate instructions is that inevitably you will end up creating infinite loops, since each instruction is randomly selected independently of previous instructions. Using the graph notation it is very straightforward to generate non-infinite loops in a controlled way. Whenever a jump instruction is selected, the length of the jump is calculated. A separate branch in the graph pads the address space between the current instruction and the jump target with instructions (jump or non-jump). In the case of conditional branches, a special rule has to be created to ensure the jump condition is not always triggered.

MODELING INTERRUPTS
A classic problem in processor verification is to ensure that certain interesting scenarios get covered. For example, will the CPU behave correctly if an interrupt occurs in the middle of a multiplication?

```verilog
prll_instr = instr1 instr2;
constraint only one mult c {
    if ( instr1 . op inside [UMUL, SMUL]
        {instr2.op outside [UMUL, SMUL] ;};
    };
```

Figure 6: Micro-architectural constraints based on pipeline resources

In the pipeline case, instr1 is the stored previous instruction, and instr2 is the one being generated. So only traversal of instr2 sends a new instruction to the testbench (a pipeline of depth N is modeled by storing N-1 instructions).
One of the useful aspects of the graph notation is that graphs can be made reactive. By using an interface as described above, a graph can read states and variables from the DUT or testbench, and modify its behaviour based on the state(s). One way to take advantage of this is to create a separate independent graph to generate interrupts. This graph would monitor the current state of the CPU, and when a specific condition occurs, like a specific instruction or sequence of instructions, an interrupt is injected.

**SUMMARY**

In this article we have shown how graphs can be used to efficiently model CPU/GPU/DSP instruction sets. A single graph can model an entire instruction set, or separate graphs can emphasize different scenarios, re-using modular sub-graphs of certain types or groupings. This approach enables efficient coverage closure of scenarios and corner-cases, e.g. combinations of opcode, source and destination, or interesting sequences of instructions fed into a pipeline.

Also, graph models are independent of language or simulation environment, and are re-usable at any abstraction level.

A demonstrator model is presented, based on the Sparc V8 instruction set.

**REFERENCES**

The SPARC Architecture Manual, Version 8, Sparc International Inc.
**On-Chip Debug – Reducing Overall ASIC Development Schedule Risk**

*by Eric Rentschler, Chief Validation Scientist, Mentor Graphics*

**INTRODUCTION**

With ASIC complexity on the increase and unrelenting time-to-market pressure, many silicon design teams still face serious schedule risk from unplanned spins and long post-silicon debug cycles. However, there are opportunities on both the pre-silicon and post-silicon sides that can be systematically improved using on-chip debug solutions. In the pre-silicon world, there exists a great opportunity for a solution that can provide more cycles of execution than what is possible today with simulation and emulation. But in the past, functions like power management span HW, FW, BIOS, virtualization and OS levels and are difficult to cover until silicon hardware is available. The most recent industry tape-out data shows that despite the use of state-of-the-art pre-silicon verification techniques, the number of spins to get to market is not only higher than planned, but is not improving over time. On the post-silicon side, the black-box complexity of today’s designs calls for enhanced internal observation. Many design teams could benefit by enhanced debug logic. A more intuitive debug GUI can enable higher productivity through quicker adoption. The Certus™ on-chip logic analyzer solution addresses much of this in a systematic manner as an off-the-shelf solution. Certus can be deployed in FPGA or in ASIC silicon, with an easy-to-use GUI for instrumenting the debug logic and interacting with the instruments in silicon. After a decades-long legacy of EDA improvements in test, verification and physical design, Mentor is bridging the gap into the post-silicon realm. This makes cross-functional teams significantly more productive and reduces schedule risk, while allowing teams to focus more on their core business.

**PERSPECTIVE: PRE- AND POST-SILICON**

The goals of pre-silicon verification are similar to that of post-silicon validation. In both cases the goals are to find and root-cause all the bugs in as little time as possible. Success relates directly back to basic control and observe, which has become more challenging over time as design size and complexity expand in step with “Moore’s Law.”

In the half century history of chip design the EDA industry has evolved around increasing specialization. Important areas of chip design that were once done by chip design teams are standardized and automated. Design teams benefit through productivity gains. Their teams stay more focused on their core business and tackle designs of more complexity without as much staffing. Noteworthy areas where this has happened include test, physical design and pre-silicon verification. The test world adopted scan insertion and ATPG software for high push-button coverage and low test times. The physical design world was revolutionized by synthesis and place and route software tools. The area of pre-silicon verification benefitted from coverage tools, directed random stimulus, assertion-based verification, code & functional coverage and more. But the area of post-silicon validation/debug has largely been left as a black art that still falls on the shoulders of the design teams with home-grown, pragmatic but incomplete and difficult to use solutions.

On the pre-silicon side, the observability is very good; everything is observable in simulation and emulation. But due to performance requirements, it is difficult to apply adequate stimulus to fully validate full chip and system-level functionality. There just are not enough cycles to run sufficient verification content, especially when testing & validation incorporate application, driver or even test software. Although emulation has helped significantly, the drive for more verification cycles in less time continues leaving silicon as the only option for faster dynamic verification. These challenges are due to many factors. The sheer size of the state-space resulting from the consolidation of functionality on a single die dictates a very limited number of SoC-level pre-silicon simulation cycles before tape-out. Increasingly complex security and power management schemes have added to the mix as well. Significant interactions between hardware/firmware/BIOS/virtualization/drivers/OS cannot be fully proven until real hardware. Power management algorithms are distributed across many of these layers, yet are a significant source of bugs.

On the post-silicon side it is possible to run lots of content. But, observation is a significant challenge as designs are too complex. For example, if the SoC power is coming in over budget, where do you start to look to narrow-down the problem?

**THE STAKES ARE GETTING HIGHER AS WELL**

With society’s increased reliance on computer systems in a broader range of applications, there’s less room for error. In the past, only the military-aerospace sector had safety critical systems. Today, we now have automated medical devices, self-driving cars, autonomous drones and control of large commercial mission-critical infrastructure. A malfunction due to a critical bug escaping to the field could result in serious injury, or death.
THERE IS A PATH FORWARD
With the advent of FPGAs above the 10 million "equivalent ASIC" logic gate size, large ASIC designs can now be partitioned across multiple FPGAs and prototyped to run at speeds of ~10 MHz or faster. At these speeds it starts to get quite interesting for running meaningful software content before tape-out. But without the native observability present in simulation or on an emulator, the validation and debug story remains the missing piece. An on-chip logic analyzer solution allows teams to root-cause complex issues in a much shorter time than is possible using any other practical means. We refer to on-chip logic analyzers as “debug engines” in this article. This applies both in an FPGA prototyping environment as well as on the actual post-silicon space.

WHAT ARE THE KEY CAPABILITIES REQUIRED IN SUCH A SOLUTION?
Debug engines have become a powerful tool for large, modern ASICs and their corresponding FPGA-based prototypes. They allow for extensible solutions that solve the black-box problem created by the high consolidation of functionality in today’s chips. They allow for full-speed observability of arbitrary state buried deeply in chip designs. They are typically deployed in a distributed manner within IP blocks across a chip. A subset of signals from an IP block is fed to its local debug engine. They support an ability to trigger on anything they can observe. They can filter and trace the observed data with the traces extracted after the run for analysis.

Debug engines can range significantly in capability. A basic debug engine can cover use cases such as:

- Trigger on an event
- Trace an event
- Count occurrences of an event

Events could be a single bit going high or could be a decode of multiple signals indicating a specific bus cycle, for example. Triggering includes both level-based as well as edge-based modes. Tracing can be to dedicated on-chip storage, shared storage or even to off-chip storage for more depth. The ability to count clock cycles, as well as a number of event occurrences is another key capability of a basic debug engine.

The capabilities above, while relatively simple, cover what's needed for a significant percentage of debug issues. However, there are less frequent but more challenging issues that beg for more capability.

In order to maximize validation and debug effectiveness at an affordable area cost, it is important to be selective about what to observe with an on-chip logic analyzer. Categories of items to connect to debug engines include:

- Errors/machine checks
- Assertions
- Performance monitors
- Interrupts
- Breakpoints
- State machines
- Flow control signals
- Queue depths
- Power state

Note that a significant number of assertions from the pre-silicon verification environment can be included in an FPGA prototyping implementation, depending upon area constraints. For ASICs the number and complexity of assertions is typically more limited. One category of challenging bugs is what has been referred to in the industry as "long-latency detect" bugs. These are bugs where the time from when something went wrong to the time when it is observed is large. In these cases, much of the state characterizing the sequence of events leading up to the original fault is long-gone. Coherency bugs frequently fall into this category.

In order to root-cause these more difficult bugs, more debug sophistication is needed. This includes features such as:

- Deep trace
- Sequential state decode
- Cross-triggering between blocks

Deeper trace is always valuable, but it is not free. In order to save on both trace bandwidth, as well as trace depth, a lossless data compression scheme is very valuable. Off-chip trace options are sometimes limited on actual ASICs due to pin constraints.
The sequential state decode feature of a debug engine allows for use cases such as: look for N occurrences of pattern A, followed by pattern B, then trigger.

Another key feature is the ability to cross trigger between debug engines in different IP blocks in order to debug over the context of the entire system. In the case of an FPGA prototype that includes multiple chips, hence cross triggering should span all the FPGAs. The cross triggering must work robustly across arbitrary clock domain boundaries and optimally, support reasonably low latency. The key use-cases enabled by this triggering include the ability for:

- Any debug engine in the system to trigger
- One debug engine observe an event and arm another debug engine to begin looking for another event.

This is very valuable when debugging across IPs. These cases are not uncommon, where different IPs are constantly being combined to make up an SoC.

Issues that straddle hardware and software are very commonplace today. These interactions can span the entire stack including application, OS, hypervisor, BIOS, firmware and hardware. For this reason it is important to have a solid hardware/software co-debug story. Being able to correlate events between hardware and software is very powerful.

These features can also be used as coverage points in validation, allowing more direct observe of items that may be difficult to infer using other techniques.

In order to achieve the promise of significant productivity enhancements, it is very important to make all these capabilities as easy as possible to use. Rather than end up with a small number of expert users, the need is to enable a broad range of users across disciplines. Users can span diverse groups including hardware, driver, BIOS and software.

One area for ease-of-use is adding the debug infrastructure to the design, aka instrumenting the design. The other big area for ease-of-use is interacting with the debug logic at run-time. Typically, there is a much larger set of users interacting with the debug logic at run-time than there are in the instrumentation part.

Mentor’s Certus debug solution offers an industry-leading set of features and can be deployed on FPGAs and ASICs. This allows users to benefit from the convenience of the same infrastructure in their FPGA prototyping environment, as well as their post-silicon environment.

Certus offers an easy to use GUI for instrumenting a design, the “Implementor”. When a user is adding debug engines and observation points to a design, debug resource utilization is reported real-time. Signal selection is also very flexible and intuitive. Users can easily navigate the design hierarchy and use filters to quickly identify the signals to make observable. Implementor operates on source RTL and generates readable, instrumented RTL. A TCL script facilitates the verification of instrumented RTL.

The Certus run-time GUI tool is called “Analyzer” and has also been tailored for ease-of-use. Since Certus separates the instrumentation of signals for observability from the selection for active tracing, it provides for fully flexible, intuitive selection of instrumented signals. Each debug engine in the system is then programmed by the Analyzer based on the active trace set selection and the desired triggering and conditional capture expressions for a specific run. After a run, Analyzer pulls trace data from the trace buffers and creates a waveform file viewable by users just like a pre-silicon simulation. RTL signal names are applied and waveforms are time-correlated, removing all tedium from the user experience.

The overall ease-of-use of the Certus tool makes user adoption fast, quickly increasing the overall productivity of teams working on root-causing issues.

By employing an easy-to-use and full-featured on-chip debug solution, FPGA and ASIC designs can benefit from reduced schedule risk, reduced number of chip spins, improved quality, reduced time-to-market, and higher levels of customer trust. Mentor is committed to continually enhancing the Certus tool capabilities for customer productivity spanning both pre and post-silicon. Additionally, by using the off-the-shelf and industry leading Certus solution, design teams can have enterprise-level support and focus significantly more on their core business, all at an affordable cost.
Hardware Emulation: Three Decades of Evolution—Part III
by Dr. Lauro Rizzatti, Verification Consultant, Rizzatti LLC

THE LAST DECADE
At the beginning of the third decade, circa 2005, system and chip engineers were developing evermore complex designs that mixed many interconnected blocks, embedded multicore processors, digital signal processors (DSPs) and a plethora of peripherals, supported by large memories. The combination of all of these components gave real meaning to the designation system on chip (SoC). The sizes of the largest designs were pushing north of 100-million gates. Embedded software—once virtually absent in ASIC designs—started to implement chip functionality.

By the end of the hardware emulation’s third decade, i.e., 2015, the largest designs are reaching into the billion-gate region, and well over 50% of functionality is now realized in software. Design verification continues to consume between 50% and 70% of the design cycle.

These trends became tremendous drivers for hardware emulator vendors to redesign and enhance their tools, opening the door to opportunities in all fields of the semiconductor business, from multimedia, to networking, to storage.

Following a decade of turbulence characterized by the sudden appearance of new players and, just as quickly, their demise, and a few mergers and acquisitions, by 2005, only three players competed in the emulation market: Cadence® Design Systems, Mentor Graphics® and EVE (Emulation and Verification Engineering), each promoting its own architecture.

CADENCE
After dropping the commercial FPGA-based approach around 2000, Cadence opted for the processor-based emulation technology. It was first introduced in 1997 by Quickturn Design Systems under the commercial name of CoBALT™ (Concurrent Broadcast Array Logic Technology). Based on an IBM technology, the processor-based emulator became one of Cadence’s showcase technologies after it acquired Quickturn in 1999. Under the new ownership, five generations of said technology were introduced with the name of Palladium®. Although Cadence does not release what process technology node has been used in the various versions, the best guess is listed in table I.

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>Palladium I</td>
<td>2002</td>
<td>90nm</td>
</tr>
<tr>
<td>Palladium II</td>
<td>2004</td>
<td>90nm</td>
</tr>
<tr>
<td>Palladium III</td>
<td>2006</td>
<td>65nm</td>
</tr>
<tr>
<td>Palladium XP</td>
<td>2010</td>
<td>65nm</td>
</tr>
<tr>
<td>Palladium XP2</td>
<td>2013</td>
<td>65nm</td>
</tr>
</tbody>
</table>

Table I. Here’s a look at five generations of Palladium beginning with its introduction in 2002, and the best-guess process node technology used in each.

While the principle of operation has been the same in all versions, each new generation expanded design capacity, improved speed of execution, added new capabilities for design analysis and design debug, and fine-tuned and accelerated the compilation process. The main advantages of Palladium have always been fast compile time and 100% design visibility at full emulation speed without compilation.

Palladium also excels in in-circuit-emulation (ICE) mode, supported by a large catalog of speed bridges. The main drawbacks have been large physical dimensions and higher power consumption when compared to an FPGA-based emulator with an equivalent design capacity. The impact of this drawback has affected scalability. Palladium-XP2 would require 32 boxes to reach the maximum capacity of two-billion gates as specified in the datasheet. Finally, in transaction-based acceleration, Palladium is reported to perform at lower speed than its competitors.

MENTOR GRAPHICS
After acquiring IKOS Systems in 2002, Mentor Graphics undertook the challenging task of merging the custom FPGA-based technology inherited via the acquisition of Meta Systems in 1996, with the Virtual Machine Works technology implemented in the IKOS emulators. The marriage of the two patented approaches parented the Veloce® emulator, launched in 2007 with the tagline of “emulator-on-chip.” Fast and easy design setup, swift...
compile time, and 100% design visibility at full speed without compilation are the hallmarks of this approach.

The custom FPGA developed for emulation does not have the capacity of the largest commercial FPGAs on the market, leading to more custom devices to map any given design size compared to using off-the-shelf FPGAs. The net result is that the emulator has larger physical dimensions and a heavier weight with longer interconnecting wires and slower propagation delays than an emulator using commercial FPGAs.

Mentor also inherited the research IKOS conducted in the transaction-based verification field before the acquisition. Repackaged and enhanced under the name of TestBench Xpress™ (TBX), it is the most effective implementation of the transaction-based verification in emulation. Mentor took the technology a step further, by implementing VirtuaLAB, comprising specialized testbenches for specific applications, such as USB and Ethernet, in an all encompassed package.

In 2012, Mentor introduced a new generation of Veloce, called Veloce2. Veloce2 retained all the advantages of the emulator-on-chip architecture, it doubled the maximum design capacity, and it significantly expanded its design analysis capabilities, adding low-power verification, a unique power estimation approach, and functional coverage. For embedded software debugging, Mentor devised Questa® CodeLink that, unlike the JTAG approach, does not seize up the emulator and it does not intrude on or interfere with the operation of the system being run.

EVE/SYNOPSYS

In the middle of the first decade of the new millennium, the merger of IKOS and Meta Systems technologies at Mentor Graphics proceeded at a slow pace. It opened the door for a startup named Emulation Verification Engineering (EVE) to play a role in this attractive market. EVE launched in 2003 its first emulation product implemented on a PC-card under the name of ZeBu®-ZV. (The name ZeBu is a contraction of zero bugs.) Unlike the custom silicon approach of Cadence and Mentor, EVE elected to use the Xilinx® Virtex®-II FPGAs as the building block for its emulator.

The pioneering product met with a discreet success, leading to a chassis-based emulation platform with vastly more Virtex-II FPGAs, called ZeBu-XL. This dual implementation approach, PC-card and chassis, became the development model for several years. Table II lists all the EVE’s emulation products and the FPGA type used in them.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Year</th>
<th>Xilinx FPGA</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZeBu-ZV</td>
<td>PCI-Card</td>
<td>2003</td>
<td>Virtex-II</td>
</tr>
<tr>
<td>ZeBu-XL</td>
<td>Chassis</td>
<td>2004</td>
<td>Virtex-II</td>
</tr>
<tr>
<td>ZeBu-UF</td>
<td>PCI-Card</td>
<td>2006</td>
<td>Virtex4</td>
</tr>
<tr>
<td>ZeBu-XXL</td>
<td>Chassis</td>
<td>2007</td>
<td>Virtex4</td>
</tr>
<tr>
<td>ZeBu-Personal</td>
<td>PCIe-Card</td>
<td>2008</td>
<td>Virtex5</td>
</tr>
<tr>
<td>ZeBu-Server</td>
<td>Chassis</td>
<td>2009</td>
<td>Virtex5</td>
</tr>
<tr>
<td>ZeBu-Server2</td>
<td>Chassis</td>
<td>2012</td>
<td>Virtex6</td>
</tr>
</tbody>
</table>

Table II. Seven generations of ZeBu, their implementation footprint, the introduction year, and the Xilinx Virtex FPGA used are shown in this table.

When compared to a configuration with the same design capacity, EVE’s ZeBu platforms included fewer FPGAs in a smaller chassis than a corresponding custom-FPGA based emulator. This characteristic led to shorter propagation delays inside the emulator achieving faster speed of execution.

The faster emulation speed came at the expense of severe limitations. Long setup and compile time were drawbacks that users had to accept to gain faster emulation. While ZeBu supported 100% design visibility without compilation using the built-in Xilinx Virtex feature called “read-back,” it did so at very low speed of few tens or hundreds of hertz.

Breaking away from the typical emulation setup, ZeBu-ZV did not support the ICE mode, focusing instead on transaction-based verification. Forging a new method, it mapped the transactors in a dedicated FPGA, called flexible testbench (FTB), and clocked at twice the speed of the design-under-test (DUT) for higher bandwidth.

ZeBu-XL and all the following products supported ICE, but the FTB approach was its priority, and became a characteristic in the DNA of ZeBu ever since.

Synopsys® acquired EVE in 2012 and, two years later, launched ZeBu-Server3 based on the Xilinx Virtex-7 FPGA. The latest generation of the commercial-based emulator boasted more capacity, lower cost and higher speed. It also
improved its compilation speed, and expanded the design analysis capabilities.

NEW DEPLOYMENT MODES AND NEW VERIFICATION OBJECTIVES

Hardware emulation was conceived from the beginning to be deployed in ICE mode to support real I/O traffic albeit at lower performance than the real speed. This was the trait for almost two decades. Everything changed at the beginning of the third decade, when the transaction-based methodology experienced by IKOS was adopted by all emulation vendors.

Today, hardware emulation is the only verification tool that can be deployed in several modes, some of which combined for added versatility. See table III.

Because of this versatility, hardware emulation can be used to achieve several verification objectives. See table IV.

Multi-user support, i.e., the ability to accommodate multiple concurrent users in one emulation platform was first introduced by Quickturn CoBALT and quickly adopted by the other vendors. The combination of multi-user and remote access capabilities gave rise to emulation data centers, actively promoted by Mentor Graphics.

<table>
<thead>
<tr>
<th>Deployment Mode</th>
<th>How to Implement</th>
<th>Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-Circuit Emulation (ICE)</td>
<td>DUT mapped inside the emulator, connected to physical target system in place of a chip or processor.</td>
<td>Fast execution speed with real I/O traffic (interfaces via speed-bridges).</td>
</tr>
<tr>
<td>Transaction-Based Acceleration (TBA, TBV or TBX)</td>
<td>Physical target system replaced by virtual target system written in C++, SystemVerilog, or SystemC, connected to DUT via transactors.</td>
<td>As fast execution speed as ICE. No need for speed adapters. Deterministic debug. Unmanned remote access. Higher reliability.</td>
</tr>
<tr>
<td>Simulation Testbench Acceleration</td>
<td>RTL testbench drives DUT inside emulator via programmable logic interface (PLI).</td>
<td>Slowest performance mode. No need to change testbench.</td>
</tr>
</tbody>
</table>

Table III. Hardware emulation includes four main modes of deployment, including two sub-modes.

<table>
<thead>
<tr>
<th>Verification Objective</th>
<th>What/Why/How to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Debugging</td>
<td>Foremost application, offering speed of execution between 100,000X and 1,000,000X faster than hardware description language (HDL) simulators. It provides an accurate representation of the design based on a silicon implementation before actual silicon is released by the foundry.</td>
</tr>
<tr>
<td>Hardware/Software Co-verification or Integration</td>
<td>It ensures that embedded system software works as intended with the underling hardware. It can trace a software bug propagating its effects into the hardware and, conversely, a hardware bug manifesting itself in the software’s behavior.</td>
</tr>
<tr>
<td>Embedded Software Validation</td>
<td>It provides the necessary processing power (in billions of cycles) for embedded software validation, from drivers, to operating systems, applications, diagnostics, and software-driven tests.</td>
</tr>
<tr>
<td>System-Level Prototyping</td>
<td>Only hardware emulation can accommodate more than one-billion gate designs and process several billion cycles without running out of steam.</td>
</tr>
</tbody>
</table>
Table IV., cont.

<table>
<thead>
<tr>
<th>Verification Objective</th>
<th>What/Why/How to</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low-Power (Power Island) Verification</td>
<td>Low-power design verification can be achieved by modeling the switching on/off of power islands and related issues, such as retention, corruption, isolation, and level shifting, as defined in a power format file (UPF for Mentor/Synopsys or CPF for Cadence).</td>
</tr>
<tr>
<td>Power Estimation</td>
<td>Emulation can track switching activity at the functional level and generate a database to be fed to power estimation tools (Cadence and Synopsys) or directly feed the switching activity to power estimation tools via an API bypassing the file generation process (Mentor).</td>
</tr>
<tr>
<td>Performance Characterization</td>
<td>Design performance characterization can be carried out by verifying the number of cycles required for completing any specific design function.</td>
</tr>
</tbody>
</table>

**CONCLUSION**

Today, hardware emulation is at the foundation of every verification strategy, and it has become a permanent fixture in SoC design. Embedded software is forcing project teams to debug hardware and develop software in parallel. With emulation, they now have the means to do it fast, efficiently and cost effectively. New emulation solutions are meeting the need and creating a new era for hardware and embedded software designs.
INTRODUCTION

The present day designs use standard interfaces for the connection and management of functional blocks in System on Chips (SoCs). These interface protocols are so complex that, creating in-house VIPs could take a lot of engineer’s development time. A fully verified interface should include all the complex protocol compliance checking, generation and application of different test case scenarios, etc.

Our tool IDesignSpec automatically generates registers and memory interfaces which can interface with all the standard bus protocols. One of the outputs from IDesignSpec product is the Bus client RTL. The generation of this IP is challenging since what gets generated is based on the specification that the user provides.

For the reliability and credibility of our design, we need to ensure that it is fully compliant with the standard protocol. Testing out such dynamic designs is a huge task since the customer can create virtually any possible specification. So it is not a static IP that needs to be verified, nor is it a configurable IP. It really is a generated IP, so the task is an order of magnitude, more complicated.

If we take a look at the AMBA® buses, AXI4Lite bus protocol has different channels for reading and writing, which are designed to work simultaneously and this gives rise to a lot of complexity in the verification of the protocol. The AMBA® AHB bus supports features required for high performance like interfaces of masters and slaves and other complex functionalities like split transactions. So verification of such behavior with constrained resources, within a very tight schedule is challenging, to say the least.

QUESTA VIP

We were fortunate enough to get access to Questa VIP (QVIP) for our design IP verification. Questa VIP provides a huge benefit to verify the DUT by plugging it into their testbench.

QVIP is easy to use and provides plug-n-play verification environment to verify the IP design and it has enabled the achievement of higher functional coverage results. It provides comprehensive test suite, which is well documented and covers different scenarios like simple read write transaction, back to back read write transaction with N cycle delays, simultaneous read and write transaction from different address channels, read and write channel with N cycle delays and so on. It helps to verify dual port and single port memory which is connected to the slave side. It also helped to verify the custom logic at the slave side.

QVIP enabled our team to verify the bus interfaces quickly and thus reducing the effort required by the engineers to develop a very comprehensive verification solution. As a result we were able to focus on other crucial tasks in our project and also maintained the quality and reliability of our design as expected by our customers.
USING QVIP FOR THE END-TO-END VERIFICATION OF IDESIGNSPEC GENERATED AXI4-LITE, APB4 AND AHB SLAVE RTL

1. It provides complete UVM verification test bench environment for verifying any AMBA® slave with sample wrappers for connecting to user’s slave and inbuilt sequences.
2. The easily understandable built-in examples in the VIP, explains different usage models like tests for simple read write, for coverage, to check backdoor access and ARM® compliance. It covers memory models with N delays and provides proper response signals for the read and write transaction.
3. By the comprehensive protocol compliance checkers in the VIP, we were able to find bugs in areas like back to back transactions to the memories, with and without delays, simultaneous read write to different channels in AXI4-lite interface.
4. It provides easily configurable test bench environment, with knobs which enable configuration of the VIP for AHB and AHB-Lite.
5. VIP has examples of interface having multiple masters and slaves with inbuilt arbitration and address decoding mechanism. The number of masters and slaves are easily configurable.
6. The built-in sequence covers all kinds of master transfers and slave responses, ensuring maximum protocol coverage.
7. The VIP monitor checker contains assertions that warn the violation of interface protocol. These assertions can also be disabled through configuration.
8. It enables application of user’s stimulus very easily through sequences, with ability to provide inline constraints.
9. QVIP allows the closed loop verification, where it provides the verification plan and has inbuilt reusable scoreboards to collect the coverage data.

A COUPLE OF THINGS THAT CAME UP WHILE USING QVIP

1. To use the AHB/APB QVIP for verifying our slave RTL, we needed to constrain the addresses generated by the master, in order to configure the slave for some specific addresses. We had to manually set the slave address range configuration in the test by using the methods “set_config_slave_end_address_range” and “set_config_slave_end_address_range”. The good thing is that in the newer version of QVIP (10.4c), slave address range is automatically set in the BFM through the system address map settings and hence we will not need to use the configuration variables to set the slave address range manually.
2. Another thing that came up is that most of the addresses in the range specified by the slave address range configuration methods as discussed above, were not hit. So we applied constraints directly to the slave address variable “rand bit [[(AHB_ADDRESS_WIDTH) - 1):0] address” in the transaction class “ahb_master_burst_transfer”. My constrained address range was crossing the 1kb boundary, which is a protocol violation. But in QVIP, constraints to prevent crossing the 1kb boundary for a burst were already applied. This led to a randomization failure. To prevent this randomization failure, we could turn off the constraint specified by the QVIP, by setting constraint_mode(0).

CONCLUSION

The QVIP enabled us to easily find buggy areas that would have been hard to reach. We rely on Mentor Graphics for its complete verification IP solutions and excellent customer support. They empowered us to give a positive proof to our customers that our generated IP is compliant with the standard.
Most false positives (i.e. missing design bugs) during the practice of model checking on industrial designs can be reduced to the problem of a failing cover. Debugging the root cause of such a failing cover can be a laborious process, when the formal testbench has many constraints. This article describes a solution to minimize the number of model checking runs to isolate a minimal set of constraints necessary for the failure. This helps improve formal verification productivity.

1. INTRODUCTION

Formal verification in the form of model checking has become an integral part of the chip design sign-off process at many semiconductor companies. The process involves building formal testbenches that can exhaustively verify certain design units. This process complements the simulation testbenches that typically run at full-chip or subsystem level.

There can be many reasons why any verification methodology (simulation or formal) can give rise to false negatives or false positives, undermining the confidence built by the verification process. False negatives are typically less worrisome because they are accompanied by a failure report or a counterexample that can be analyzed to determine if the root cause is a design bug. False positives in formal verification are important to eliminate, and often debugging the root cause of such false positives can be very tedious.

This topic assumes an even higher importance since formal verification is a newer methodology, and the reasons for false positives are different than in simulation. One of the main reasons for false positives is presence of over-constraints, often resulting in constraints that unexpectedly conflict with each other. The focus of this article is to make it easier to debug such false positives.

The first place to start is to identify the presence of false positives. Some symptoms of false positives are:

- A user-written cover property fails unexpectedly
- A tool-generated cover on a sub-formula of a user-written assert fails (typically reported as a vacuous proof by a formal tool)
- The proof time for an assert is much faster than expected!
- Dead-ends are reported by either the simulation or the formal tool

For the remainder of this article, we will assume that the formal user is adept at identifying such symptoms and reducing them to a failing cover on the design, although this process in itself can be non-trivial.

Debugging the root cause of a failing cover can be a hard problem, especially in formal testbenches that consist of hundreds or thousands of constraints. In all situations we have ran into, such a cover also fails in the presence of a small subset of these constraints, typically between 2 to 5. Once such a subset has been identified, we have usually been able to relate this subset to an unintentional mistake made by the formal engineer. In this article, we describe a method to minimize the number of model checking runs to identify a minimal failing subset (MFS) of constraints.

We have run into the problem of debugging vacuous proofs often, and most formal tools do not provide this solution automatically. Therefore it is important that formal users know how to solve the problem manually with minimum effort.

To solve the problem, we find the MFS by running the model checking tool multiple times, after modifying a set of asserts, covers or constraints. We will call each such run a step, with the goal of trying to minimize the number of steps. Assuming that the initial failing cover does not incur a large run-time, likely each or most of such steps finish fast too. Nevertheless, since each step requires some human time, we would like to avoid hundreds or thousands of runs, if the number of constraints n is large, i.e. a solution that takes O(n) steps is an unacceptable solution to our problem.

2. DATA PATH CONTROL DESIGN

To show an application of this problem, we show a simplified version of a real verification example. Consider a design (Fig. 1, shown on the following page) consisting of two source ports A and B, and a single destination port. Note that the interfaces for the A and B are different from each other.
2.1 Design Functionality

The design arbitrates between the two sources under a strict priority scheme of arbitration, i.e., source A is serviced at higher priority than source B. In order to back-pressure the data coming from the sources, the design uses the ready_a output signal on source A interface and the wait_b output signal on source B interface. There is a state machine in the design (Fig. 2) that governs which source is being serviced. The outputs ready_a and wait_b remain asserted as long as design arbiter is in SERVICE_A state. Likewise, the output wait_b remains asserted.

2.2 Formal Verification Testbench

The formal verification testbench consists of a set of constraints and a set of checkers as shown in Fig. 3. The testbench contains the following checkers on the outputs of the design:

- A1: Data correctness checker that data sent for source A must match the expected data ref_data_a, modeled in a scoreboard:

  \[ \text{source_a_sel && valid_out |-> (data_out == ref_data_a)} \]

- A2: Data correctness checker that data sent for source A must match the expected data ref_data_b:

  \[ \text{source_b_sel && valid_out |-> (data_out == ref_data_b)} \]

In addition, the testbench contains the following constraints on the inputs of the design:

- C1: Once asserted, valid_a should remain asserted until ready_a is asserted

  \[ \text{valid_a && !ready_a |-> ##1 valid_a} \]

- C2: If valid data is sent from source A, it should hold its value until ready_a is asserted

  \[ \text{valid_a && !ready_a |-> ##1 (data_a == $past(data_a))} \]

- C3: If wait_b is asserted, then valid_b should be deasserted in the next clock cycle

  \[ \text{wait_b |-> ##1 !valid_b} \]

- C4: If valid_b is deasserted, then start_b should also be deasserted

  \[ !valid_b |-> !start_b \]

- C5: If valid_b is deasserted, then end_b should also be deasserted

  \[ !valid_b |-> !end_b \]
\!valid_b \implies \!end_b

- C6: start_b and end_b should be mutually exclusive
\!\!(\text{start}_b \&\& \text{end}_b)

- C7: If valid_b and wait_b is asserted, then valid_b should hold its value in next clock cycle
\text{valid}_b \&\& \text{wait}_b \implies \#1 (\text{valid}_b == \$\text{past} (\text{valid}_b))

- C8: If valid_b and wait_b is asserted, then data_b should hold its value in next clock cycle
\text{valid}_b \&\& \text{wait}_b \implies \#1 (\text{data}_b == \$\text{past} (\text{data}_b))

It is not hard to get the tool to report why the checker A2 was passing vacuously. In fact, it was easy enough to reduce that problem to the surprising result that the following cover was failing (i.e. the design was never sending any data out for source B!)

\text{witness}_{\text{src}_b_{\text{data}}_w}: \text{cover property (}
@\text{posedge clk) disable iff (rst)}
\text{source}_b_{\text{sel}} \&\& \text{valid}_{\text{out}}
\text{)};

### 2.4 Minimal Failing Subset of Constraints

The debug of the vacuous proof in the previous section forced the engineer to identify a minimal subset of constraints that is sufficient to cause the cover to fail. In fact, once she solved that problem, she realized that of the hundred of constraints, C3 and C7 were sufficient to produce the unexpected failing cover. At this point, a manual review of these two constraints was feasible. And she realized that C3 and C7 conflict with each other because in state SERVICE_A, the output wait_b is asserted, and if valid_b is asserted here, C3 will force valid_b to be deasserted in the next clock cycle, whereas C7 will force valid_b to be asserted in the next clock cycle. C3 was the incorrect constraint. And it happened because at some point in the project, she had realized that her understanding of the source B interface was incorrect and needed to be modified to implement that by C7 and C8. But she had forgotten to remove C3 on that interface, which was based on her earlier incorrect understanding!

The need to debug failing covers happens enough times during formal verification, and the debug of that is so tedious that we were motivated to solve this problem more efficiently.
3. PROBLEM AND SOLUTIONS

We are given a model checking tool that can verify whether a cover passes or fails on a given design under any set \( S \) of \( n \) constraints (the cover is said to fail if the model checking tool proves the cover is unreachable, and it is said to pass if the tool finds a witness to reach the cover from reset). To avoid verbosity, if the cover passes, we will say \( S \) PASSES; conversely, if the cover fails, we will say \( S \) FAILs. The goal is to find any minimal FAILing subset (MFS) of \( S \) using as few steps as possible.

If no MFS is larger than \( m \) (note we do not know \( m \) beforehand), the problem is called a problem of size \((m; n)\). (Note that \( 1 \leq m \leq n \).)

The following solution returns a minimal \( T \subseteq S \) such that \( T \cup S_{aux} \) FAILs. To find an MFS of \( S \), we simply run the algorithm with \( S_{aux} = \emptyset \).

![Diagram of solution process]

**Fig. 4 and 5 illustrate some example cases.**

A similar algorithm to solve this problem was published earlier by Junker\(^1\). Both Junker’s algorithm and ours have the same worst-case complexity \( O(m \log n) \). We implemented both his algorithm and ours, and the results appear in Table 1. When there is only one MFS, i.e. there is only value of \( m \), both Junker’s algorithm and our algorithm...
take similar numbers of steps. However, when there are multiple MFSes, especially when they are disparately sized, our algorithm runs in far fewer steps than his. This is because his algorithm is biased towards finding an MFS which contains lower-numbered constraints even when such an MFS may be much larger than an MFS that contains only higher-numbered constraints. This puts his algorithm at a disadvantage.

Table 1. Results with n=1000, and 100000 trials

<table>
<thead>
<tr>
<th>m</th>
<th>Ours</th>
<th>Junker</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>15.01</td>
<td>15.02</td>
</tr>
<tr>
<td>{2}</td>
<td>27.55</td>
<td>27.55</td>
</tr>
<tr>
<td>{3}</td>
<td>39.11</td>
<td>39.11</td>
</tr>
<tr>
<td>{4}</td>
<td>49.93</td>
<td>49.94</td>
</tr>
<tr>
<td>{10}</td>
<td>106.56</td>
<td>106.58</td>
</tr>
<tr>
<td>{1, 1}</td>
<td>14.52</td>
<td>14.52</td>
</tr>
<tr>
<td>{1, 2}</td>
<td>16.60</td>
<td>16.63</td>
</tr>
<tr>
<td>{1, 10}</td>
<td>15.06</td>
<td>22.96</td>
</tr>
<tr>
<td>{2, 2}</td>
<td>26.49</td>
<td>26.81</td>
</tr>
<tr>
<td>{2, 3}</td>
<td>30.02</td>
<td>31.42</td>
</tr>
<tr>
<td>{2, 10}</td>
<td>28.83</td>
<td>40.18</td>
</tr>
<tr>
<td>{1, 2, 10}</td>
<td>16.64</td>
<td>20.74</td>
</tr>
<tr>
<td>{1, 10, 100}</td>
<td>15.06</td>
<td>23.85</td>
</tr>
</tbody>
</table>

4. CONCLUSION

The article discussed a solution that requires \( O(m \log n) \) steps of model checking runs to find the minimal FAILing subset of constraints; since typically \( m << n \), this is useful. This solution has increased our productivity by reducing the iterations required to debug vacuous proofs.

REFERENCES

A Generic UVM Scoreboard
by Jacob Andersen, CTO, Kevin Seffensen, Consultant and UVM Specialist, Peter Jensen, Managing Director, SyoSil ApS

All UVM engineers employ scoreboarding for checking DUT/reference model behavior, but only few spend their time wisely by employing an existing scoreboard architecture. The main reason is that existing frameworks have inadequately served user needs, and have failed to improve user effectiveness in the debug situation. This article presents a better UVM scoreboard framework, focusing on scalability, architectural separation and connectivity to foreign environments. Our scoreboard architecture has successfully been used in UVM testbenches at various architectural levels and across models (RTL, SC). Based on our work, the SV/UVM user ecosystem will be able to improve how scoreboards are designed, configured and reused across projects, applications and models/architectural levels.

I. MOTIVATION; EXISTING WORK

Addressing the increasing challenges met when performing functional verification, UVM proposes a robust and productive approach for how to build and reuse verification components, environments and sequences/tests. When it comes to describing how to scoreboard and check the behavior of your design against one or more reference models, UVM offers less help. UVM does not present a scoreboard architecture, but leaves the implementer to extend the empty uvm_scoreboard base class into a custom scoreboard that connects to analysis ports. Experience shows that custom scoreboard implementations across different application domains contain lots of common denominators of deficiency. Users struggle to implement checker mechanisms for the designs under test being exposed to random stimuli, while sacrificing aspects like efficiency, easy debug and a clean implementation.

Existing user donated work suggests some UVM scoreboard architectures, offering UVM base classes and implementation guidelines together with some examples of use. These implementations commonly require the user to write an “expect” function that is able to check the design under test (DUT) responses once these transactions arrive at the scoreboard. The use of such a non-blocking function imposes numerous modeling restrictions. Most notably, only one model can be checked, namely the DUT. Secondly, it is difficult to model many of the parallel aspects of common DUT types, leading to messy implementations where the comparison mechanism becomes interwoven with the queue modeling. The “expect” function may be suitable for rather simple applications (e.g. packet switching), but does not fit more generic use cases. Lastly, self-contained SystemC/TLM virtual prototypes are difficult to incorporate as “expect” functions in such scoreboard architectures.

We find existing proposals inadequately address our requirements for a state-of-the-art scalable scoreboard architecture. Therefore we have created a scoreboard implementation that has been used across multiple UVM verification projects, and we would like to share our experiences and the guidelines we have set up. For the UVM community to benefit from our work, our scoreboard library has been released and is available for download (see section IX).

II. SCALABILITY & ARCHITECTURAL SEPARATION

Our scoreboard is able to simultaneously interface and compare any number of models: Design models (RTL, gate level) and timed/untimed reference models (SystemVerilog, SystemC, Python). As a logical consequence, we insist on a clear architectural separation between the models and the scoreboard implementation, the latter containing queues and comparison mechanisms, and we specifically choose not to employ the “expect” function concept.

To simplify text and schematics, the sections below explain the architectural separation between a DUT and a reference model (REF), tailored to check the DUT running random stimuli. In subsequent sections, we will present how the scoreboard (SCB) interfaces to other models, and compares the operation of any number of those.

A. Division of Tasks; REF vs SCB

A REF typically implements a transaction level model of the RTL DUT, written in SystemVerilog, C/C++, SystemC or similar languages, and may be inherited from system modeling studies. The abstraction level of the model would typically be “PV - Programmers View” but any of the well known abstraction levels (“AL - Algorithmic Level”, “PV - Programmers View”, “PVT - Programmers View with Timing” and “CA - Cycle Accurate”) can be used. A transaction level model does not model exact RTL pipeline
and timing characteristics, and for some stimuli scenarios the order of transactions on the interfaces might differ between the models.

For each DUT pin level interface, the REF will have a transaction level interface. These interfaces might transfer information in and out of the device (e.g. a read/write SoC bus protocol), or only transport information in a single direction (e.g. a packet based protocol). Depending on the exact goal of the complete verification environment, the reference model might model the full DUT functionality, or only parts of it. This depends on the ambitions for running random simulations and what practically is possible to model in the transaction level reference model. For instance, a reference model of a packet switch might model the packet flow, but refrain from modeling the control flow (credits), as this would require the model to be fully or partially cycle accurate.

The SCB does not model the DUT. It merely queues the transactions going in and out of the DUT and the REF, and is able to compare the activity of the models, using a specific algorithm for comparing the data streams. This algorithm might range between a precise in-order and a relaxed out-of-order compare, depending on how well the reference model is DUT accurate. Methods implementing such comparison algorithms are a standard part of the SCB base class layers. Custom algorithms may be capable of analyzing individual transactions, and put forward more specific requirements for transaction ordering. Such custom algorithms are easy to implement and deploy in the SCB framework.

B. Implementation Details

Figures 1 and 2 present the structure and interconnect of the REF and the SCB. Here, we have multiple REFs to show how the solution scales. The DUT is the primary model. This is determined by the DUT being attached to the verification environment that creates the design stimuli with UVM sequences. The verification components (VCs) drive the DUT stimuli, and from the VC analysis ports, transactions (UVM sequence items) are picked up by one or more REFs (M1 … Mn) as well as the SCB. The REFs are trailing secondary models, as these are unable to proceed with execution before the primary model stimuli have been created, applied, and broadcast by the VC monitor. The REFs are connected to the VCs using analysis FIFOs, allowing the REF to postpone handling the transaction until potential dependencies have been resolved on other ports.

![Figure 1. DUT, trailing models and scoreboard architecture. Bi-directional information flow on DUT ports. Configuration suitable for SoC bus interfaces.](image)

When considering a SoC bus interface with the VC as initiator, the transactions received by the REFs contain both a req part (request, DUT stimuli) and a rsp part (DUT response), as we do not expect the UVM environment to offer an analysis port sending only the req part. The REF needs the req part, and will replace the rsp part with a computed response, based on the req and the REF internal state. For instance, the req part can be a bus read request (address and byte enables), whereas the rsp part then will be the data read from the DUT. It is mandatory for the REF to create its own transaction (e.g. by using the sequence item copy method), as altering the transaction received from the analysis port will also alter the transaction received by the SCB from the DUT.
For each model (M1 ... Mn) attached to the scoreboard, any number of SCB queues can be handled. Each queue contains meta-transactions, wrapping UVM sequence items along with metadata. This allows a single queue to contain different transaction types not necessarily comparable with each other. The metadata is used to ensure that only queue elements of the appropriate type are compared.

The queues can be organized in several ways. The SCB in the figure displays the normal configuration; one queue per model, with different transaction types in the queues. The generic compare methods only compare transactions of the appropriate type (A, B) between the queues. Organizing one single queue per model is simple to configure, and provides great help when debugging failing scenarios, as a timeline with the full transaction flow can be visualized. The alternative queue configuration shown in the figure is one queue for each port for each model. For some application types it might be desirable to configure the queues in this fashion, if the relationship between the ports is irrelevant for debugging the SCB queues.

The generalized REF/SCB configuration in Figure 1 can be simplified to Figure 2 for the special case where the DUT ports only employ a uni-directional information flow, e.g. a packet based traffic pattern. Neither DUT/Mx response on the A ports nor testbench stimuli on the B ports occurs. Hence the structure resembles more traditional scoreboard usage, where a DUT packet stream simply is compared to the REF packet stream. In the figure we show how the stimuli (A port) still can be added to the SCB queues. This will not add value to the performed checks, but will greatly aid the debug situation, as the queues will present both input and output DUT/REF traffic.

To summarize, the presented scoreboard architecture is capable of handling both uni- and bi-directional port traffic. Furthermore, the SCB is fully separated from one or more trailing reference models, allowing the use of REFs with port topologies matching that of the DUT, and potentially modeled in different languages/domains than SystemVerilog. Also, the separation promotes reuse, e.g. by reusing module level SCBs at the SoC level.

III. NON-UVM CONNECTIVITY

Besides interfacing to UVM using analysis ports, establishing links to non-UVM/non-SystemVerilog code is essential to keep the scoreboard versatile and reusable, enabling the use of external checkers and debug aiding scripts. For this purpose, the scoreboard framework offers a number of external interfaces:

The portable UVM Connect library enables seamless TLM1/TLM2 communication between SystemVerilog/UVM and SystemC. We employ the library for implementing most run-time interfaces listed above. For connecting analysis ports between SV/SC, we employ uvmc_tlm1 sockets with generic payloads, using the “sv2sc2sv” pattern where SV and SC both act as producer/consumer. To use this pattern, pack/unpack methods must be available on both sides of the language boundary. Today we use the UVM field macro generated pack/unpack methods in SV, and the UVMC macros for the SC side. If performance issues arise in a specific implementation, a shift to using the dedicated UVM pack/unpack macros is implemented.

To allow external resources to receive a transaction stream, we offer interfaces both in SC/C++ and Python. These interfaces are mainly intended to be used at run-time, such that external scripts are being evaluated while the SV simulation.
is running, avoiding creating large log files for post-simulation processing.

For the Python App Socket, the user has to manually implement the pack/unpack methods on the Python side. Once done, the user can write Python scripts with analysis ports, where a function is called every time a sequence item is received by the SCB. Hence the user can attach extern Python scripts with analysis capabilities not present in SystemVerilog – either if too difficult to model – or if existing Python analysis scripts are available. Furthermore Python is an easy bridge towards other tools, where run-time streaming of SCB activity is needed.

Table 1: Non-UVM Interfaces.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Language</th>
<th>Execution</th>
<th>Purpose/Benefits</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP</td>
<td>SystemC</td>
<td>Run-time</td>
<td>Reference model (virtual prototype)</td>
</tr>
<tr>
<td>Queue</td>
<td>SystemC/C++</td>
<td>Run-time</td>
<td>Use for high performance queue comparison, low memory footprint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Use for interface to existing C/C++ protocol checkers</td>
</tr>
<tr>
<td>App Sockets</td>
<td>Python</td>
<td>Run-time</td>
<td>Use for creating checkers, easily creating complex data structures and debug aiding scripts</td>
</tr>
<tr>
<td></td>
<td>Custom</td>
<td>Run-time</td>
<td>Use for interfacing to any language callable from C++</td>
</tr>
<tr>
<td>Logger</td>
<td>XML / TXT</td>
<td>Post-sim</td>
<td>Streaming socket to log file, XML and/or TXT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Usable for post-simulation analysis purposes</td>
</tr>
<tr>
<td>Non-UVM</td>
<td>SV</td>
<td>Run-time</td>
<td>Interface to non-UVM SystemVerilog interfaces (not depicted in the figure below)</td>
</tr>
</tbody>
</table>

IV. IMPLEMENTATION & CONFIGURABILITY

The SCB implements a generic scoreboard architecture, which is organized in any number of queues. In turn, each queue is able to contain items from different producers, indicated by tagging with different meta data. This forms a two-dimensional structure with MxN entries (refer to Figures 1 and 2).

For configuring the SCB, a configuration class offers a dedicated API, making it easy to configure the number of queues and producers. For populating the SCB with traffic from the DUT and REFs, two different APIs can be used to insert items into the queues:

- Transaction based API: Insertion by using the automatically generated UVM analysis exports. Used for the normal UVM use cases.
- Function based API: Insertion by calling the add_item method. Used if transactions require transformation before insertion, e.g. if not extended from uvm_sequence_item, or if hooking up as callback. See section V for example.

Both mechanisms automatically wrap a given uvm_sequence_item with metadata and insert it into the given queue. Thus, there can be any relationship between the number of ingoing transaction streams and the number
of queues and item producers in the scoreboard. It is up to the implementer to choose a meaningful use for the queue and item producer concepts, while keeping in mind that all pre-packaged compare methods delivered with the SCB architecture aim to compare all elements of corresponding item producer types across all queues.

The SCB evaluates the contents of the queues whenever a new item is inserted, using the configured method to remove matching entries that satisfies the criteria of the compare method, as well as reporting whenever elements in the queues violate the compare rules. The performance of the compare is directly tied to the performance of the queue implementation. Thus, only an abstract queue API description is defined along with several implementations.

Models produce transaction flows in different orders. The scoreboard allows specification of the evaluation criteria, triggering when a comparison between a set of queues is evaluated. This is done by promoting a specific queue to become a primary queue, leaving the secondary queues to be trailing. A time consuming RTL model will typically be set as the primary model, whereas a zero-time transactional reference model will be set as a secondary queue. Furthermore, evaluation will also be triggered at end-of-simulation to ensure that all queue contents are as expected.

The generic scoreboard architecture is implemented by extending standard UVM base classes. This allows us to use the UVM Factory to specialize a scoreboard implementation, e.g. by changing the comparison algorithm for a specific test. In Figure 4, the UML diagram of the scoreboard UVM classes is shown. For more information about the implementation, refer to5.

The SCB comes with several “ready to use” compare methods as listed in Table 2. The default comparison strategy is in-order by producer. If the compare methods provided by the scoreboard do not fit the required compare scheme, then a custom compare can be implemented by extending the cl_syoscb_compare_base class. For easing the implementation, each queue has a locator object attached which provides a collection of search algorithms for traversing the queues in an elegant and easy manner. Additionally, standard iterator objects are also available for iterating through either search results or selected parts of a queue.

V. COMPREHENSIVE EXAMPLE
Below right—and on the following pages—a comprehensive code example is supplied, showing how to declare, build, configure and connect the scoreboard.

Figure 4. UML diagram: Scoreboard Classes.
<table>
<thead>
<tr>
<th>Compare Method</th>
<th>Description of Compare Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Out of Order (cl_syoscb_compare_ooo)</td>
<td>Performs a 1:1 element out of order compare across all queues. Used to check that each secondary queue contains same contents as the primary queue, but without any specific ordering restrictions. When a matching set is found, elements are removed from the respective queues. Error reporting: If a queue grows above a set threshold, or a queue timeout happens.</td>
</tr>
<tr>
<td>In Order (cl_syoscb_compare_io)</td>
<td>Performs a 1:1 element in order compare across all queues. Used to check that each secondary queue contains same contents as the primary queue, and that the ordering is exactly the same, also regarding the producer types. When a matching set is found, elements are removed from the respective queues. This will always be the first element of both primary and secondary queues. Error reporting: If the first element in a secondary queue is different from the first element in the primary queue, disregarding the producer type. Also if a queue threshold/timeout happens.</td>
</tr>
<tr>
<td>In Order by Producer (cl_syoscb_compare_io_producer)</td>
<td>Performs a 1:1 element in order compare across all queues. Used to check that each secondary queue contains the same contents in the same order as the primary queue but only within the same producer. Thus, this is less strict than the normal in order compare. When a matching set is found, elements are removed from the respective queues. This will always be the first element of the primary queue. Error reporting: If the first element in a secondary queue of a specific producer type is different from the first element in the primary queue of the same producer type. Also if a queue threshold/timeout happens.</td>
</tr>
</tbody>
</table>

Table 2: List of prepackaged compare methods.

```perl
import pk_syoscb::*;

class cl_scbtest_env extends uvm_env;
    cl_syoscb     syoscb; // Handle to SCB
    cl_syoscb_cfg syoscb_cfg; // Handle to SCB config
    myagent       agent1; // Handle to agent1
    myagent       agent2; // Handle to agent2

    `uvm_component_utils_begin(cl_scbtest_env)
    `uvm_field_object(syoscb_cfg, UVM_ALL_ON)
    `uvm_component_utils_end

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

        // Use the standard SV queue implementation as scoreboard queue
        cl_syoscb_queue::set_type_override_by_type(cl_syoscb_queue::get_type(),
                                                  cl_syoscb_queue_std::get_type(),
                                                  **);
```

www.mentor.com	Mentor	www.mentor.com
31
// Set the compare strategy to be OOO
factory.set_type_override_by_type(cl_syoscb_compare_base::get_type(),
  cl_syoscb_compare_ooo::get_type(),
  "**");

// Create the scoreboard configuration object
this.syoscb_cfg = cl_syoscb_cfg::type_id::create("syoscb_cfg");

// Configure the SCB (Two queues: ‘Q1’ and ‘Q2’ with a single producer)
this.syoscb_cfg.set_queues("Q1", "Q2");
void(this.syoscb_cfg.set_primary_queue("Q1");
void(this.syoscb_cfg.set_producer("P1", {"Q1", "Q2"});

// Pass the scoreboard configuration object to the config_db
uvm_config_db #(cl_syoscb_cfg)::set(this, "syoscb", "cfg", this.syoscb_cfg);

// Create the scoreboard
this.syoscb = cl_syoscb::type_id::create("syoscb", this);

// Create and configure agent1 and agent2
...
endfunction: build_phase

// Use the TLM API to add SCB subscribers to the monitors of
// agent1 and agent2
function void connect_phase(uvm_phase phase);
  super.connect_phase(phase);

  begin
    cl_syoscb_subscriber subscriber;

    // Get the subscriber for Producer: P1 for queue: Q1 and connect it
    // to the UVM monitor producing transactions for this queue
    subscriber = this.syoscb.get_subscriber("Q1", "P1")
    this.agent1.mon.ap.connect(subscriber.analysis_export);

    // Get the subscriber for Producer: P1 for queue: Q2 and connect it
    // to the UVM monitor producing transactions for this queue
    subscriber = this.syoscb.get_subscriber("Q2", "P1")
    this.agent1.mon.ap.connect(subscriber.analysis_export);
  end
endfunction: connect_phase
endclass: cl_scbtest_env
VI. DEBUG
Most scoreboards just echo the difference between the expected and the actual transaction flow. Our scoreboard architecture offers mechanisms for understanding the full transaction flow and model context at the point of failure. The scoreboard offers two forms of debugging aid:

Logging: During normal simulations, the scoreboard keeps down the queue sizes by evaluating the instantiated compare method. When an error happens, the remaining queue content is written to the simulation logs, displaying the “diff” of the queues at the point of failure. With this information, it is difficult to diagnose the cause of the error post-simulation, as the output does not contain the full simulation history. By enabling the “full scoreboard dump” feature all elements added to the scoreboard queues during the simulation will be dumped to a set of files. The dump mechanism dumps to text format or XML format depending on the configuration of the scoreboard. Also, XLST transformations can convert the transaction dumped XML into other file formats, e.g. GRAPHML XML to produce timed transaction graphs.

Analysis: Using the APIs described in section III, external verification scripts can get run-time access to the transaction streams, aiding the debug process by analyzing the streams and producing higher-order views of the traffic. A good example would be to visualize the enumeration process on a PCIe bus system. Analysis scripts can be implemented to read the dumped XML file or access the transactions at runtime, exploiting rich data structures in external languages. This by far is a better solution than implementing scripts that employ Unix greps or regular expressions.

VIII. IMPROVEMENTS
Since first published\(^6\), the underlying implementation of the UVM scoreboard has undergone some changes in order to improve the usability and tool compatibility with the three big simulator vendors. In general three major changes have been done:

- Changes in the class hierarchy.
- Different methods for enforcing APIs.
- Minor changes to obtain simulator compatibility.

For more details, refer to the scoreboard documentation.\(^5\)

IX. GENERAL AVAILABILITY
Our UVM Scoreboard architecture has been released for general availability, and can be downloaded from the following web resources:

- Accellera UVM Forum\(^5\)
- SyoSil homepage

The release features the UVM Scoreboard base classes, examples, release notes and documentation.

The scoreboard has been released under the Apache 2.0 license, and can be used freely. Any suggestions for how to improve the base classes and examples are very welcome, including potential bug reports. Please direct such feedback per email to the authors at scoreboard@syosil.com.
X. FUTURE EXTENSIONS
SySi is committed to continue the development of the UVM Scoreboard. We currently plan to include the following extensions:

- Additional implementations of compare methods, including examples of how to employ special rules taking the contents of the UVM sequence items into consideration.
- Additional queue implementations, optimized with better locators.
- More configuration “knobs”.
- A general mechanism for communicating side effects from the reference model to the scoreboard.
- Even better debug aiding mechanisms.

XI. CONCLUSION
In this work we propose an industry-proven, scalable UVM scoreboard architecture, able to interface to any number of design models across languages, methodologies and abstractions. Any relationship between data streams can be checked using pre-packaged and custom compare methods, and we make it easy to interface external checker and debug aiding applications. Based on our work, the SV/UVM user ecosystem will be able to improve how scoreboards are designed, configured and reused across projects, applications and models/architectural levels.

XII. REFERENCES
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

25 Video Courses Available Covering
- Formal Verification
- Intelligent Testbench Automation
- Metrics in SoC Verification
- Verification Planning
- Introductory, Basic, and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- PowerAware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 6000 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com