It's cold here in New England this winter. We had a nasty ice storm just before Christmas that uprooted a large tree at the edge of my backyard. We were fortunate that the tree fell in between the shed and the swing set, damaging neither, but I was still left with the problem of trying to clear all of this mess over the Christmas break. Fortunately, I'm blessed with very helpful neighbors — one of whom has a chainsaw and a pickup truck — and my son, David, who just turned 11. It took two days to cut away all the brush, which David dragged across the yard and piled next to the driveway and then my neighbor helped me load it into his truck for the two trips to the dump it took to dispose of it all. There's still more to be done, but at least now it can wait until it gets warmer.

As I sat down to write this note to you, I was thinking of the ice storm as a metaphor for the economic downturn we've been experiencing. The current economic climate is as inhospitable as a New England winter, so to survive we have to “bundle up” and try to keep warm. In business terms, that means keep going as best we can while minimizing our exposure to risks. With that thought in mind, and in the spirit of being helpful neighbors, this issue of Verification Horizons will show you how Mentor Graphics can help you reduce risk across your verification processes.

“In the winter in New England, it’s important to stay warm and be thankful for friendly neighbors and family to help in times of need. In a stormy economy, it’s even more important to protect yourself from the “cold” of risk in your verification process.”

—Tom Fitzpatrick

Our feature article for this issue comes from our friends at Contemporary Verification Consultants, a Questa Vanguard partner located in India. In “Achieving a Higher Quality Design through Questa,” they present a case study detailing the advantages of many of the advanced verification techniques discussed in previous issues.
of Verification Horizons. The article shows how constrained-random stimulus, functional coverage, and the SystemVerilog DPI, particularly as supported by Questa®, were used to more efficiently create a more robust verification environment than the previous directed-test environment. Their focus on functional coverage allowed them to exercise many more scenarios beyond what the designers could code by hand, thus exposing many issues early in the process that had been missed in the previous design. The more robust verification environment resulted in a higher quality product using a much more predictable, and less risky, process.

Our next two articles deal with minimizing the risks associated with verifying the increasing complexity of your designs. In “Reduce the Risk of Expensive Post-Silicon Debug,” you’ll see how our 0-In® CDC product can accurately model the metastability inherent in your multi-clock designs — you know, those intermittent bugs that usually show up only in the lab. The article shows you how those bugs can now be found and eliminated as part of your RTL verification step. Next we look at another type of synchronization in “Mitigate Multi-Processor Synchronization Risks with Processor Driven Verification.” The article discusses the requirements for an effective multi-processor verification environment, especially in synchronizing the processors with each other and providing a powerful non-intrusive debug environment that shows each processor and all of the hardware together. You’ll see how Questa Codelink™ makes it easier to debug and verify such a complex SoC design.

Another area where many of our customers are looking to reduce risk is when moving to the Open Verification Methodology (OVM) from their current legacy Verification Methodology Manual (VMM) environments. While they recognize the advantages of the OVM (as discussed in previous Verification Horizons articles), there are times when an engineering team is reluctant to replace or rewrite legacy VMM components, so interoperability has become a critical issue. In December, Mentor Graphics released an open-source interoperability library to address this problem (see http://www.ovmworld.org/contributions-details.php?id=30). “Reusing Legacy VMM VIP in OVM Environments” gives an overview of this solution, which addresses the requirements set out by the Accellera Verification IP Technical Subcommittee.

In our Partners’ Corner, you’ll find the third and last installment of Doulos’ “A Practical Guide to OVM” series (see the June and October 2008 issues for parts one and two). This part of the guide shows how to get started writing sequences in OVM 2.0. I’d like to take this opportunity to thank John Aynsley and all our friends at Doulos for contributing this valuable multi-part guide. We conclude this issue with our Consultants’ Corner article, contributed by Mentor’s European Verification Practice Consulting group, on using the SystemVerilog “bind” construct to provide a low-risk migration path to incorporate advanced verification techniques in HDL-based testbenches.

So, in winter in New England, it’s important to stay warm and be thankful for friendly neighbors and family to help in times of need. In a stormy economy, it’s even more important to protect yourself from the “cold” of risk in your verification process. Remember that we here at Mentor Graphics will be ready with a “chainsaw” and a “pickup” to help you weather the storm.

So now it’s time to sit by the fire, grab a cup of hot chocolate, and settle in to enjoy this latest issue of Verification Horizons.

Respectfully submitted,

Tom Fitzpatrick
Verification Technologist
Mentor Graphics
# TABLE OF CONTENTS

**Page 5... Achieving Higher Quality Design through Questa**  
*by Ajeetha Kumari, CEO and V. Govind Bagath Singh, Verification Lead, CVC*

**Page 11... Reduce the Risk of Expensive Post Silicon Debug**  
*by Rindert Schutten, Senior Marketing Manager, Mentor Graphics*

**Page 14... Mitigate Multi-Processor Synchronization Risks with Processor-Driven Verification**  
*by Marc Bryan, Product Marketing Manager, System-Level Engineering Division, Mentor Graphics*

**Page 18... Reusing Legacy VMM VIP in OVM Environments**  
*by Tom Fitzpatrick & Adam Erickson, Verification Technologists, Mentor Graphics*

**Partner’s Corner:**

**Page 22... A Practical Guide to OVM—Part 3**  
*by John Aynsley, CTO, Doulos*

**Consultant’s Corner:**

**Page 27... Are You in a “bind” with Advanced Verification?**  
*by Virginie Mallez, Mark Peryer, Andy Walton, and Paul Williams, Verification Practice, Mentor Consulting (Europe)*
“The current economic climate is as inhospitable as a New England winter, so to survive we have to “bundle up” and try to keep warm. In business terms, that means keep going as best we can while minimizing our exposure to risks.”
The design and verification of a complex image processing system consisting of various image enhancement techniques is a very challenging and daunting task. The sheer complexity of algorithms and their wide range of applications mean that there are many different scenarios and corner cases to verify and also to ensure that the system is behaving correctly. Traditionally such designs have been verified with mostly directed tests and teams have been successful in doing so. However, with more complex systems being designed with shrinking time-to-market, this approach doesn't scale up. The complexity factor requires more powerful verification technologies while the time-to-market pressure involves managing and mitigating risks. Recent advances in Higher Level Verification techniques aim to address these challenges. However, a raw set of features such as constraints, coverage, assertions etc. alone don't appeal directly to such design teams as they are left to ponder which one to use where. This is where a tailored verification approach based on proven and robust verification platform fits in.

In this paper, we share our recent experience of verifying an image correction algorithm block using sophisticated verification techniques offered by SystemVerilog along with robust and easy-to-use verification platform - Questa.

**DUT DESCRIPTION**

An image captured via CCD sensor contains known defects and a piece of hardware attempts to correct these errors on the fly. The defects may appear anywhere in the picture and the CCD manufacturers provide a list of known-bad-pixel locations along with the CCD. Depending on the quality and the cost of the CCD, the defect density varies. Several algorithms exist to correct such defects with almost all of them involving replacing the defective pixel with a "computed" value. The complexity of the algorithm varies depending on the expected defect density profiles. Whereas a previous generation design used a simple algorithm that looked at adjacent pixels, the new design involved a more sophisticated one that involves four adjacent pixels and four diagonal pixels. Care should be taken to use only good pixels from the neighbourhood while computing a replacement value; for instance if there are two adjacent rows with defective pixels, the computation should skip immediate row values and those shall not be used in the computation.

The functionality of the block is best explained with the following figures. In these two sample images, the LHS one shows the defects (marked in BOLD for ease-of-identification) and the RHS is the corrected image. The defective locations shown are only indicative with the real-life ones being more dense and wide spread.

**PREVIOUS DV APPROACH**

In the previous project, a similar verification was done using Verilog based testbench with the image being fed in as input and the defects fed in via testcases. Code coverage was used to measure the implementation level completeness. It worked and found design errors; the kind of defect density profiles that were simulated was directly dependent on the creativity of the verification engineer. The directed tests missed critical corner cases as they tend to revolve around a few known weak areas where defects were more likely. With no access to unlimited DV resources, the designs were taped-out after a rigorous series of directed test runs. The resultant designs met a class of applications such as casual photography, but the quality of design was...
below acceptable standards for some other target applications, such as professional photography.

THE DV SCRUTINY

The DV team along with the system architects did a methodical scrutiny of the DV process to find areas for improvement for the next project. As a valued QVP (Questa Vanguard Partner) member and SystemVerilog based verification services provider, CVC (www.noveldv.com) was invited to the review. During the scrutiny process the following key observations were made:

• The defect density profiles verified were limited to the DV engineer’s imaginations/timelines.
• Reliance on single metric of implementation coverage masked certain spatial defect occurrences.
• Lack of self-checking ability caused inefficiencies in the entire process leading to several golden files, at times stagnant from previous runs.

The DV team wanted to look at ways to improve the quality standards and mitigate the risk in the new project that involved even more complex algorithm for correction.

THE DV SAVIOR

With the key weak areas identified, the CVC team was tasked with quickly coming up with solutions to address the issues. One additional requirement enforced was that any new approach should be widely usable and deployable across the Design and Verification team. The teams were quite familiar with ModelSim and hence they preferred a GUI that’s as friendly (or better). After a careful analysis of the DV scrutiny results, CVC suggested a modern verification approach involving three new technologies to address the previous shortcomings:

• Apply constrained-random technique to create the defect density profiles
• Add functional coverage on top of code coverage
• Build a self-checking mechanism via SystemVerilog DPI

While some of the DV team members quickly appreciated the new proposal, others were apprehensive about the potential added risks such as:

• Constrained- random is not easy to measure/visualize/track
• Teams need to analyze new metrics, probably learn new coverage analysis tools
• Integration difficulties of previous generation C-Verilog interface – PLI

As a QVP member, CVC addressed these concerns by doing a quick demo of the Questa verification platform and SystemVerilog. The key takeaways of that demo were:

• Questa’s solver is indeed robust and provides a wider distribution to cover various corner cases. When augmented with a well-crafted functional coverage model, the randomness can be measured and progress indicators can be quickly generated.
• The functional coverage analysis in Questa is nearly the same as the code/implementation coverage metric that the teams were familiar with. Intuitively laid out coverage results made most of the analysis self explanatory.
• A few members wanted to take the combination to the next level by the ability to combine implementation coverage with functional coverage. With Questa’s advanced Unified Coverage DB capabilities this was achievable (though yet to be deployed).
• SystemVerilog’s DPI is a higher abstraction, functional-level interface than detailed signal-level one. This meant that the system architect’s C-golden reference model can directly be reused during the simulation phase to qualify the test run as PASS/FAIL.

REVAMPED DV SAVOIR FAIRE

Adopting Functional Coverage

Our team added functional coverage on existing design and showed why the code coverage alone (used in previous project) was not sufficient and how it missed some important corner cases and scenarios. One of the characteristics of an image processing block is its data centric, computation heavy implementation. The old verification was focused only on getting all the computation logic intact, but it overlooked the spatial occurrence of the defects at various possible spots. There were many directed defects inserted and verified but the verification completeness was only as good as the individual creativity of scenarios. Unfortunately code coverage, which showed full coverage for all the computational elements such as adders, shifters etc doesn’t truly reveal functional holes, which were missed. Once the...
functional coverage mindset was introduced the designers and micro-architects came up with interesting coverage goals. The table shows a few of them.

<table>
<thead>
<tr>
<th>Coverage aspect</th>
<th>Cover expression</th>
<th>Comments</th>
<th>Preview</th>
</tr>
</thead>
<tbody>
<tr>
<td>Two consecutive rows of defects</td>
<td>Bad_pixel_loc</td>
<td>Computation should ignore “known bad pixels” and use the “corrected values” or skip the bad row</td>
<td></td>
</tr>
<tr>
<td>Intersecting defect rows &amp; columns</td>
<td>Bad_pixel_loc</td>
<td>Intersections get corrected twice</td>
<td></td>
</tr>
<tr>
<td>Is every pixel location marked as defective (across sim runs)</td>
<td>Bad_pixel_loc X image_dimension (Cross)</td>
<td>Ensure that the correction algorithm can handle defects at any location</td>
<td></td>
</tr>
</tbody>
</table>

As the samples above indicate, functional coverage brings in higher abstraction than low-level implementation coverage; one could appreciate the “visualness” in the defect pixel location. SystemVerilog provided the necessary language hooks to capture such advanced coverage goals at a higher level of abstraction. Questa provides an intuitive, easy to comprehend view of the coverage results very similar to its code coverage analysis, which made the verification engineers feel at home while analyzing new, more complex information. As the team became more and more familiar with functional coverage, they learned how much more useful it is to digest and analyze the information it provided vis-a-vis the raw data that code coverage used to provide.

**Exploiting constraint solver capabilities**

The next challenge was to ensure the full set of coverage goals was achieved in a reasonable time frame. While the textbook definition of constraints tends to project as if constraints shall be used to model all stimuli, our team took a rather pragmatic approach. We used the raw image inputs and used constraints to randomize the defect locations and their density. With the powerful solver engine in Questa, very interesting defect density profiles were randomly generated and fed to the DUT. Such profiles are very hard, if not impossible, for directed testcase writers to conceive and code. As with any new technology, early users of constraint-random verification are prone to make errors in their constraint models. Questa comes with a nice, easy-to-use solver debug interface that points quickly to the source code level problem. A few useful tips for the early constraint users are:

- Use `–solvefailedebug` flag with `vsim` command line. This prints extended debug information regarding constraint conflicts and also tries to create a reduced testcase to understand the constraint model in isolation.
- As the model grows bigger, using the command line option may not be the most preferred mechanism. This is especially true where multiple folks are concurrently developing parts of testbench code. Questa provides a nice attribute that can be in-lined in the randomize() call that provides solver debug for selected parts of testbench alone.

```plaintext
class def_dens_gen;
    function void generate();
        success = def_profile.randomize (*solvefailedebug*);
    endfunction
endclass
```

One of the significant benefits of adopting a constrained random testbench is the ease of adding additional tests compared to a traditional, directed testbench. With the old testbench setup, every time a new testcase is needed, the de-facto approach is to copy-paste the old test and tweak the defect density profiles. With constrained random testing there are very significant changes:

- First of all, using different random seeds is very likely to create different defect density profiles. In Questa one can use the `vsim –sv_seed` option to pass a different seed. With this, the same SystemVerilog test code creates different “density profiles”. This is very powerful as it eliminates the need to maintain several tests in the old setup.
- While using the random seed is a good option, the ROI (Return on Investment) of running several seeds tends to taper off at some point. This is not a bad sign, but rather an indication of how constrained the state space is. SystemVerilog provides various ways to alter the state space targeted by constrained random generation such as:
- Turn constraint blocks ON/OFF
- Inline constraints to further shrink the state space
- Layered constraints to override base class constraints etc.

In the above code snippet, a derived defect density profile is inherited from the base class and additional constraint has been added to focus on the corners of the image. With a layered testbench setup, it is quite easy to swap the random generator (a verification component within the environment) blueprint with an extended version as shown above. The power of SystemVerilog combined with a layered approach to testbench building results in significant line count reduction compared to older testbench styles for similar requirements. In our system we could easily measure 2X to 3X difference in line count reduction depending on the type and scale of modifications needed in the testcase.

Another visible result of using constrained random generation along with a well crafted functional coverage model is the impact on productivity in achieving these results. Assuming that we added the functional coverage alone to the old setup, it would take a significant number of additional, directed/focus tests to achieve all the coverage goals. It also spoils the true motivation behind the coverage driven verification flow in some ways: the coverage points are “directions” to look for potential faulty areas in design and not necessarily “directed targets”. If it is the latter, a set of directed tests is sufficient and some argue that the effort in writing the coverage model is redundant. The real benefit of coverage driven verification is harvested if the coverage points are used as “directions” instead. That is, the testing should seek “around” these areas rather than trying to hit exactly these spots. This is quite hard to do with the old setup. However with the new setup, as the defect profiles are randomly generated, the path to achieving coverage is to focus/constrain “around” the goals more than “exactly on the goals”. A few guidelines to keep in mind while closing in coverage holes are:

- Avoid equality constraints (== operator) – they tend to mimic a directed test
- Use “inside” operator leniently
- Prefer declarative constraints over the procedural constraints (such as in-lined randomization) – this keeps the scope for future constraints via inheritance.

Reusing reference models in self-checking

One of the common aspects of any image processing application is the availability of golden reference model (albeit at a higher level abstraction) in C/C++ from the system architects. During hardware verification of such algorithms, it is often redundant to recode the algorithm as a checker; instead teams prefer reusing the reference model. In the past such checking was mostly limited to a post-simulation step as the nitty-gritty’s of integrating a C/C++ model with Verilog through PLI is painful, time consuming and required maintenance. With SystemVerilog DPI this task has become very simple and, at times, trivial – by simply looking at the SystemVerilog code that uses the DPI calls one may not even notice that the checking function/utility is in a foreign language such as C/C++.

For instance, the code snippet shown below is that of our checker integrating a reference model from C via DPI. The function `im_ref` is actually a C-function, however unlike in old PLI/VPI, there is no “$” symbol involved here, and the integration is highly simplified.
The real challenge in using DPI is to find the right mapping of SV to C datatypes. In this project we used static, unpacked arrays across the SV-C boundary and the mapping was done as follows:

<table>
<thead>
<tr>
<th>SV datatype</th>
<th>C-datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>bit [31:0] pixel_in</td>
<td>const svBitVecVal* pixel_in</td>
</tr>
<tr>
<td>[0:307200]</td>
<td></td>
</tr>
<tr>
<td>int no_of_bad_pixel</td>
<td>const int no_of_bad_pixel</td>
</tr>
<tr>
<td>bit [18:0] bad_pixel[0:99]</td>
<td>const svBitVecVal* bad_pixel</td>
</tr>
</tbody>
</table>

The choice of 2-state data type across the boundary is a conscious one as the reference model is at a higher level of abstraction, unaware of X/Z of hardware. We used SystemVerilog’s $isunknown system function at the boundary and simple assertions to catch any violations of this assumption. The C-side function prototype looked like:

```c
#include “svdpi.h”
void im_ref(const svBitVecVal* pixel_in,
            const svBitVecVal* bad_pixel,
            const int no_of_bad_pixel,
            svBitVecVal* pixel_out)
```

Questa’s DPI integration capabilities ensured that the checker integration is smooth. Once we generate a shared object file, `vsim –sv_lib` is all that is needed to integrate it.

Questa provides an integrated C-debugger that is very handy during the debug phase of this project. It is much like standard GDB for C but with a tighter integration to traditional HDL debug windows. This made the RTL team and the Verification team feel at home while debugging foreign language. For the System team to debug the reference model, CVC demonstrated a nice way to quickly focus on C-code without having to go behind several HDL/SV lines of code. This feature is called “Auto BP” in Questa’s C-debug flow. When enabled, it sets breakpoints automatically at every SV-C boundary call without the user having to know the location of files, libraries etc. Below is a screenshot of how Questa quickly takes the user to the relevant C-entry point after “Auto BP”.

**SUMMARY**

Putting it all together, the functional coverage, constraint-random generation and the DPI were the key SystemVerilog features that proved vital for achieving a higher quality design. The power of functional coverage enabled the micro-architects to capture more stringent quality requirements as executable verification goals that were previously handled in ad-hoc manner. The combined strengths of code and functional coverage used together resulted in finding more design errors than with the previous approach of...
relying on code coverage alone. Adoption of constrained random generation brought down the number of directed tests significantly and also enabled the team to achieve tougher verification goals in a shorter span of time. Clearly the old approach doesn’t scale up well to rising demands of higher quality designs. Finally the debug of a failing simulation run got accelerated multi-fold by the reuse of the C-reference model and Questa’s well integrated debug solution. As a near future enhancement the team is exploring extending the DPI integration to every pixel level computation to isolate issues very close to their occurrence.

Needless to say the language is only as powerful as an implementation and its ease of use. Questa has excellent support for the IEEE 1800 SystemVerilog standard and its intuitive debug and analysis capabilities make advanced verification fun indeed!

ABOUT CVC

CVC (Contemporary Verification Consultants Pvt. Ltd. http://www.noveldv.com) is a high end DV Startup Company focused on THE most challenging aspect in today’s semiconductor industry - Functional Verification. We have all it takes to make the functional verification of your ASIC/SoC successful - be it:

• Defining your next generation verification environment
• Revamping your existing one
• Auditing your verification strategy and suggesting next steps or
• Adding extra resources at that crucial point in your design cycle

CVC is a member of Questa Vanguard Program (QVP) and provides SystemVerilog and OVM based consultancy to mutual customers in Bangalore. CVC offers corporate and educational trainings in areas such as VHDL, SystemVerilog, SVA, PSL, VMM, OVM etc.

Contemporary Verification Consultants Private Limited
1777/42/1, GANGOTRI, Ground & 1st Floor,
26th B Main, 40th A Cross,
Jayanagar 9th Block,
Bangalore
Karnataka, India 560069
Tel & Fax: 91-80-41495572, Email: info@noveldv.com
A 2007 study by Far West research found that less than 30 percent of ASIC designs have first silicon success, and the majority of these flawed chips have functional flaws. Hence over 70 percent of all ASICs—and anecdotal evidence suggest that for FPGAs this number is much higher—require extensive debugging in the lab, often referred to as post-silicon debug.

Post-silicon debug is hard and expensive. It requires expensive instrumentation, and observability (when compared to simulation) is strongly impaired, because only a few of the thousands of important internal signals are directly accessible during normal chip operations. As a result, post-silicon debug is often a labor intensive process.

There is, however, a category of bugs that are even harder to find and debug post silicon than “regular” functional bugs. These bugs manifest themselves intermittently and are not repeatable. These bugs seem to appear randomly, sometimes, for example, they only appear (or disappear) after the chip has warmed up. Other times they seem to happen every now and then in a completely random fashion. These bugs behave this way because they are caused by stochastic events deep inside the silicon. Locating and exterminating these bugs—compared to regular, repeatable functional bugs—is even more daunting, and a lengthy, unpredictable post-silicon debug process is almost a given. A large set of bugs in this category originate from faultily designed clock domain crossings.

Clock domain crossings are present in many chips because today’s chips (both ASIC and FPGA) increasingly require a set of asynchronous clock domains, and the communication between these clock domains, the clock domain crossings (CDC), have to be designed very carefully to ensure consistent, correct behavior in silicon. Errors in the design of these CDCs are typically not found during simulation, and therefore, if bugs are present, they escape to silicon, causing the tremendous headache of debugging intermittent bugs post-silicon. This article explains how to eliminate these bugs from appearing in silicon in the first place, drastically reducing the risk of an unpredictable, and lengthy, post-silicon debug process.

When data is transferred from one clock domain to the other designers cannot avoid violating the basic design rule that says “thou shall not violate setup/hold conditions for flip-flops.” Indeed, this rule goes on, explaining if you change the input data while inside the set and hold window around the clock edge of a flip-flop the result is non-deterministic. In fact, the output signal will be metastable, meaning that it takes an unpredictable amount of time for the output to settle. Moreover, the logic value that the output will ultimately settle to cannot be predicted.

To account for this behavior, designers need to use dedicated design structures, called synchronizers, to make sure that these metastable signals do not contaminate the rest of the design. In other words, synchronizers confine the scope of metastable signals to the internals of the synchronizers. There are various kinds of synchronizers. A basic synchronizer is a 2DFF, as shown in figure 1. For most implementation libraries, a 2DFF synchronizer reduces the probability that its output will become metastable to practically 0 (in the lifetime of the device). In [1] more background information on metastability is given.

Irrespective of which synchronizers you use, the basic non-determinism resulting from metastability is not eliminated. In fact, all synchronizers, while doing a good job of containing metastable signals, exhibit non-deterministic delays. This means that, when metastability conditions on the first flip-flop are present, you cannot accurately predict at what cycle, in the receiving clock domain, a value change will be propagated.

These metastability effects are not modeled by regular RTL or gate-level simulations. At best, what a simulation can do is check whether setup/hold violations are present and produce an X on the output; virtually X-ing out the complete simulation, so the functional impact of the non-determinism on the delay can not be analyzed. Static timing does not help either. The best it can do is warn the user that a timing violation occurs. Since this violation cannot be eliminated (because the
two clock domains are asynchronous by design), the best the designer can do is suppress the warning. So other solutions are needed to analyze the impact of this non-determinism introduced by CDC signals and prevent bugs from escaping to silicon.

It creates a behavioral metastability model for each CDC that, when linked into the simulator, mimics accurately the metastability behavior as it happens in silicon.

Although many errors are caught by this thorough analysis process, the more intricate CDC bugs can be revealed only through thorough simulation with CDC assertions and behavioral metastability models included in that simulation.

During the simulation phase, designers use their existing testbenches that they have used to verify the functional behavior of the design. By including the CDC assertions and behavioral metastability models, they automatically get a coverage report that shows how well their existing testbenches exercise the corner cases required to expose CDC bugs. It is fairly common that the initial coverage they achieve is low, meaning that the tests were not designed to create these corner cases. So it is typical that extra tests are required to create them. In doing so, bugs are often exposed and, consequently, fixed before the design is committed to silicon.

Debugging in the simulation phase of the 0-In CDC verification solution falls back on the regular debugging environment that comes with the simulator. Once you have reached your target coverage goals on the CDC assertions and behavioral metastability models, you are done and have substantially reduced the risk of CDC bugs slipping into your silicon, dramatically lowering the risk of an expensive post-silicon debug phase for your product.

Fig. 2. The first step in comprehensive CDC verification is an analysis step. In this step all CDCs are found and checked. During this step the CDC protocol assertions and behavioral metastability models are generated as well. These assertions and models are then subsequently used in step 2 — simulation. By analyzing the coverage results, designers will typically add new tests to reach the required coverage goals.

One such solution is provided by Mentor Graphics 0-In CDC offers a complete solution to identify all CDC signals and to accurately model the non-deterministic effects of metastability in your current simulation environment. It is a two-phased process: analysis followed by simulation. It comes with a dedicated GUI to analyze the results and debug the problems.

The analysis phase is a straightforward process — just read in the RTL code of the block (or complete design) and the tool automatically identifies the clock domains in your design, finds all the CDC signals, and identifies all the synchronizers in the design. No testbench is required, so designers typically use this early in their RTL design process to make sure that all the appropriate CDC synchronizers are in place. For each synchronizer, it also checks whether the design adheres to the required CDC protocols that ensure loss-free data transfer between the clock domains. In some cases it can, through formal analysis of the design, prove that the design is correct, for others it will generate a set of assertions that subsequently are used in the simulation phase to verify that the design behaves correctly.

Finally, it creates a behavioral metastability model for each CDC that, when linked into the simulator, mimics accurately the metastability behavior as it happens in silicon.

Although many errors are caught by this thorough analysis process, the more intricate CDC bugs can be revealed only through thorough simulation with CDC assertions and behavioral metastability models included in that simulation.

During the simulation phase, designers use their existing testbenches that they have used to verify the functional behavior of the design. By including the CDC assertions and behavioral metastability models, they automatically get a coverage report that shows how well their existing testbenches exercise the corner cases required to expose CDC bugs. It is fairly common that the initial coverage they achieve is low, meaning that the tests were not designed to create these corner cases. So it is typical that extra tests are required to create them. In doing so, bugs are often exposed and, consequently, fixed before the design is committed to silicon.

Debugging in the simulation phase of the 0-In CDC verification solution falls back on the regular debugging environment that comes with the simulator. Once you have reached your target coverage goals on the CDC assertions and behavioral metastability models, you are done and have substantially reduced the risk of CDC bugs slipping into your silicon, dramatically lowering the risk of an expensive post-silicon debug phase for your product.

Fig. 3. Some examples of coverage results. With 0-In CDC all the entries are generated automatically. Then, during simulation, the coverage on CDC protocol assertions and the behavioral metastability models are measured. Unless full coverage is reached on each of them, you still run the risks of CDC bugs escaping the verification process.
For many products, schedule predictability is very important. Clearly, missing bugs during the verification process decreases schedule predictability. However, one class of bugs specifically, the bugs related to clock domain crossings, can throw a wrench into an otherwise carefully crafted schedule for product launch, because these bugs manifest themselves as intermittent bugs in hardware and are extremely hard to find and debug in a post-silicon, on the bench, debugging process. The 0-In CDC verification solution is focused on finding and fixing these bugs before tapeout or before the RTL code is downloaded into the FPGA to, if not eliminate, drastically reduce the probability of these bugs entering your silicon in the first place. More information on comprehensive CDC verification can be found in [2].


Multi-processor synchronization techniques are extensions of well established single processor, multi-threaded, software based synchronization techniques. These multi-processor synchronization techniques require a high level of concurrent visibility of both hardware and processor instruction logic. The risks of effective verification of multi-processor synchronization hardware and processor instruction logic can be best mitigated using a processor driven verification methodology and supporting tools. The stimulus must come from the processor in conjunction with the system level test bench. Debug tools must be non-intrusive and provide concurrent visibility of the hardware and processor state of all processors in a multi-processor design.

WHAT IS SO HARD ABOUT MULTI-PROCESSOR SYNCHRONIZATION?

Although the design challenges of multi-processor synchronization could fill multiple issues of Verification Horizons or a whole book, let’s take a quick look at the problem to see the acute need for flawless functionality of the hardware and software synchronization logic.

There are various, well established techniques used to synchronize multiple threads running on a single processor. These are built into multi-threaded operating systems today.

- Message passing between threads is deterministic by proper construction of the operating system.
- Priorities are used to mutually exclude threads from one another.
- If the threads are cooperating effectively and mutual exclusion or interrupt masking is used, the threads preserve the data integrity of shared memory and avoid deadlock.

All these single processor thread synchronization techniques are based on software events and are built into the operating system. There is no need of specialized hardware for single processor thread synchronization.

The techniques for synchronizing threads in a single processor do not work when translated directly to a multi-core design. Only one thread can execute at a time on a single processor. Multi-processor synchronization has additional challenges because the processors are running concurrently.

- Message passing between processors is not deterministic.
- The multi-processor system is susceptible to race conditions.
- Priorities cannot be used to mutually exclude operations on separate processors.
- Neither cooperative multi-threading nor disabling interrupts can ensure data integrity in shared memory.

In a multi-processor system, synchronization requires both hardware and instructions running on the processor. Locked bus cycles for a single processor’s exclusive use of the bus, hardware semaphores for synchronizing access to shared system resources, and other inter-processor synchronization hardware support the synchronization processor instructions in the operating system. Naturally, the synchronization hardware and processor instructions must be bug free to ensure proper and reliable synchronization between the cores. Synchronization bugs are transient, sometimes non-deterministic, and generally infrequent. Problems typically occur in a tiny window of time when instructions on two or more processors execute a co-incident action such as updating a list pointer or a shared buffer size.

Engineers are always ready for a challenge, but the project and product risks of not effectively verifying the hardware and processor instruction synchronization logic in a multi-processor system are significantly greater than those in a single processor system. It may not be possible to use processor code to work around a hardware defect in the multi-processor synchronization hardware logic that gets into your silicon.

So, mitigating the multi-processor synchronization verification risk is essential for the success of your project and product.

MULTI-PROCESSOR SYNCHRONIZATION FOR A SHARED RESOURCE

Let’s look at an example of multi-processor synchronization of a shared system resource. Figure 1 shows a high level multi-processor system architecture in which the I/O is shared by both the CPU and the DSP. In this system, the Custom Logic contains some hardware semaphore logic, which is used to mutually exclude each of the processor’s access to the shared I/O. This will prevent system errors when more than one processor accesses the I/O at the same time.
From the programmer's view, a processor must first acquire the hardware semaphore before accessing the I/O. Each processor executes a single read instruction of the hardware semaphore followed by a test of the semaphore value. If the processor sees that the semaphore is set, then the processor continues on to execute the I/O procedure. Once the I/O work is done by the processor, the processor executes a single write instruction to the hardware semaphore to clear it and make it available to other processors in the system.

From the hardware engineer's view, the hardware semaphore logic contains an atomic, uninterruptible read-modify-write operation so that when the semaphore is read by any processor, the semaphore will become set only if is already clear. It is possible to avoid additional bus overhead by processors executing spin-lock instruction sequences while trying to gain access to the hardware semaphore. The hardware semaphore logic can keep track of the processors requesting the semaphore and provide an interrupt to the next waiting processor when the semaphore is cleared.

We must verify that the hardware semaphore logic works correctly with the processor instructions controlling access to the shared I/O resource. Since the hardware semaphore is also a shared resource, we must ensure that the state changes of the semaphore and the read/write operations produce the correct values in order to prevent deadlock or corrupted I/O. Deadlock could occur if both the CPU and the DSP read the hardware semaphore and received the value indicating that the semaphore is in use. Corrupted I/O could occur if both the CPU and the DSP read the hardware semaphore and received the value indicating that the semaphore is available.

**Multi-processor Synchronization Verification and Debug Requirements**

Now that we’ve looked at the problem and risks, let’s take a look at the requirements to solve the problems and mitigate the risks.

**Processor Driven Verification**

The verification stimulus to the hardware needs to match the behavior and timing of activity generated by the processor. The best way to do this is drive the stimulus from the processor by executing processor instructions on a fully functional processor model. The instructions can be simple directed tests, diagnostics, device drivers, or complete test benches written in C or assembly.

Please see the Processor Driven Verification article in volume 4, issue 3 of Verification Horizons for a more in-depth discussion of processor driven verification.

In the hardware semaphore shared resource example, we need a set of processor tests running on all of the processors which exercise the hardware semaphore logic including individual and concurrent reads and writes of the hardware semaphore across the system bus by both processors. Because the CPU and DSP are likely running at different clock speeds, these tests will need to coordinate their concurrent execution by flags in the dual ported memory. So, before we can verify the hardware semaphore logic, we must also verify the concurrent accesses by both processors to the dual ported RAM.
Non-Intrusive Debug

Because multi-processor synchronization is highly dependent on the timing relationships between hardware and processor instructions, the debug tool for hardware and processor instructions must not disturb the timing. If the debug tool disturbs the timing relationships, the tool may inadvertently mask certain bugs. These bugs, which disappear or change characteristics when one attempts to study them, are known as “Heisenbugs”.

In our shared I/O resources hardware semaphore example, the timing of signals on the bus and in the hardware semaphore logic must not be disrupted by our debug tool. When both the CPU and DSP attempt to access the hardware semaphore for the shared I/O resource, there will be bus contention and arbitration. We need to be able to see the undisturbed signals at the bus interface of the hardware semaphore in order to verify that the bus arbitration logic does not cause problems with the hardware semaphore. For example, if there are concurrent reads of the hardware semaphore, we need to verify that the hardware semaphore logic correctly sets the semaphore for the processor that was granted access to the hardware semaphore and correctly returns the set/available value to that processor. Likewise, we need to verify that when the other processor’s read executes that the hardware semaphore correctly returns the clear/unavailable value.

Unfortunately debug monitors (such as a resident RTOS process communicating with a software debugger), inserting “break” instructions, and some JTAG connections alter the real time execution and timing of the system.

Non-intrusive debug solutions include on-chip trace modules such as the ARM Embedded Trace Macrocell and the MIPS PDTrace, software simulation in a RTL or gate level logic simulator, or complete logic emulators. All of these provide some level of support for memory and register state changes plus an initial state for memory and registers. From the initial state and the log of changes, the programmer’s view of the state of the processors can be reconstructed for any point in time (within the time range of the log/trace).

Concurrent Visibility of All Processors

Naturally, one must be able to see the state of all processors at the same time in order to have a complete view of the processors. In our shared I/O resource hardware semaphore example, we must be able to the concurrent programmer’s views (primarily registers and instructions) of both the CPU and the DSP.

IS THIS ENOUGH?

So far, so good. We’ve got stimulus coming from instructions running on the processors and established the key debug requirements to view the processor activity on all cores concurrently without changing the timing relationships.

However, non-intrusive, concurrent, multi-core debug of processor driven tests is necessary, but not sufficient for effective debug of multi-processor synchronization. These tools provide the necessary processor instruction and register views. To complete the debug solution one also needs the view of the hardware logic. The complete solution requires full processor instruction debug completely correlated to the hardware inter-processor synchronization logic.

In the hardware semaphore example, we must be able to see the register and instruction state of both processors correlated directly with the signals in the hardware semaphore logic.

This solution exists today in the Questa Codelink product.

QUESTA CODELINK

Questa Codelink provides full software debug visibility and run control of RTL and gate level processor models including registers, memory, source and disassembly code, variables, and the stack. The processor debug views are available concurrently for each processor in a homogeneous or heterogeneous multi-processor design.

The processor debug views are 100% correlated with the waveforms in Questa. The processor run controls include run (forward and backward), stop, step (forward and backward), and breakpoints.

This combination of processor debug views and run controls enables the easy correlation of inter-processor synchronization events with the hardware and processor instruction synchronization logic.

Questa Codelink is non-intrusive and requires no hardware or software changes. It runs with the same fully functional processor model you already have in your design.

Although Questa Codelink provides interaction with the RTL and gate level simulator while the simulation is running, this is generally not efficient due to the length of time required to wait before viewing the execution of a few instructions on the processor. Questa Codelink facilitates efficiency in your verification process by enabling post-simulation debug. During simulation time Questa Codelink logs changes in the processor’s general purpose registers. The logging
process has less than a 1% impact on simulation time. This small delay is more than made up by the time savings of the highly interactive, post-simulation debug when the verification engineer can quickly step forward and backwards through the processor instructions and hardware waveforms and set hardware and software breakpoints as needed. One can traverse the log of a 15 hour simulation in 5 seconds.

Turning to our hardware semaphore example, we would use the completely synchronized processor register, memory, instruction views with the hardware semaphore logic waveform views to provide full visibility to every bus access of the hardware semaphore. Both the processor debug and hardware waveform views are needed for hardware semaphore accesses by any or all of the processors that are executing the processor driven test bench. Furthermore, if our hardware semaphore logic queues semaphore requests and provides processor interrupts when the semaphore is granted to a processor, we can use Codelink to view the interrupt control logic signals of each of the processors and the processor register and instructions during the execution of the interrupt handling routines.

CONCLUSION

Functional verification of multi-processor synchronization hardware logic and processor instructions is most effective using processor driven tests and non-intrusive debug tools that provide concurrent visibility of the hardware and processor state of all processors in a multi-processor design. The risks of not discovering multi-processor synchronization bugs are greater than undiscovered bugs in single processor, multi-threaded, software synchronization. The Questa Codelink product mitigates these risks by providing all the required verification and debug features for verifying multi-processor synchronization.
OVERVIEW

The Open Verification Methodology (OVM) provides users with a proven methodology for creating modular, reusable verification components and testbenches that accelerate the verification task. With more than 12,000 downloads and 5200 users on ovmworld.org, the OVM has taken the industry by storm since its open-source release just over a year ago. Having been architected specifically to encourage (enable/support) reuse from the block to system levels and from project to project, the OVM provides the ideal level of flexibility and automation to simplify the creation of verification intellectual property (VIP). Clearly, verification teams who have taken a look at the OVM have liked what they’ve seen.

Even users of the older Verification Methodology Manual (VMM) have shown substantial interest in OVM. This led to the forming of Accellera’s Verification Intellectual Property Technical Subcommittee (VIP-TSC), chartered with standardizing a solution that enables interoperability between OVM and VMM. In response to the VIP-TSC’s approval of a set of interoperability requirements, Mentor Graphics released an open-source OVM/VMM interoperability library to meet these requirements. The library includes a set of new OVM-based components and classes to handle synchronization and communication between the OVM and VMM. It also includes an enhanced version of the open-source VMM release to provide IEEE 1800 compliance and additional infrastructure that supports interoperability. It also includes an extensive set of examples and HTML-based documentation.

INTEGRATION REQUIREMENTS

The VIP-TSC approved a set of interoperability requirements on December 3rd, 2008. The solution described here includes a library and methodology that meets or exceeds these requirements. The VIP-TSC requirements describe two levels of interoperability:

• **Interconnected Model:** The VIP integrator must take into account that the underlying VIP comes from two different libraries. Successful integration thus requires knowledge of both OVM and VMM. This is the level of interoperability at which the requirements were specified.

• **Encapsulated Model:** The details of the legacy library (in this case, VMM) must be hidden from the user of the integrated VIP. The OVM wrapper will be implemented according to the interconnected model, which requires knowledge of VMM; however users of the OVM-wrapped component will only use OVM.

To be considered a successful integration of legacy VMM in an OVM environment, it must be possible for the OVM user to instantiate, control, and augment the functionality of the unmodified VMM component from the OVM testbench. Specifically, it must be possible to:

1. Instantiate the VMM component in an OVM environment (or component) and have it run with its phases synchronized to the appropriate OVM phases.
2. Extract transactions from the VMM environment to be converted and passed to OVM components for scoreboarding, coverage, or other analysis.
3. Configure the VMM component from an OVM environment or test.
4. Specify additional transaction sequences in OVM, utilizing existing VMM transactors.

The remainder of this article will discuss how the OVM/VMM interoperability library addresses these requirements.

PHASE SYNCHRONIZATION

Both OVM and VMM support the basic concept of “phases,” but each has its own specific set of phase methods. In order for IP from the two libraries to work together, the appropriate phases from each must be called at the appropriate point relative to the other. Ideally, this alignment should be transparent to the user. The interoperability library introduces a new component, the ovm_vmm_env, which serves as a proxy to allow the OVM phase controller to control the phasing of VMM environments by registering the VMM phase methods as additional custom phases in the default phase list. The relative phasing is shown in Figure 1, with OVM phases shown in green and VMM phases shown in pink.
The library enhances the vmm_env class to include an instance of the ovm_vmm_env class, which contains a set of virtual methods corresponding to the vmm_env class' phase methods. When the OVM phase manager calls a phase method of the ovm_vmm_env, the proxy automatically calls the corresponding method of the vmm_env. As shown in the figure, the vmm_env class' start() and wait_for_end() methods are called directly from the proxy's run() method, while stop() and cleanup() are called from the proxy's stop() method, which is spawned from run(). To avoid a naming conflict, the proxy implements a reportvmm() method, which calls the vmm_env class' report() method.

The fact that the ovm_vmm_env proxy is built into the vmm_env class allows a vmm_env class to be instantiated directly in an OVM test (or other ovm_component), without modification as shown here:

```
module top;
    class vmm_tb_env extends vmm_env;
        ... endclass
    class test extends ovm_test;
        vmm_tb_env env;
        function void build();
            env = new();
        endfunction
    endclass
    initial
        run_test("test");
    endmodule
```

The vmm_env class gets instantiated in the test's build() method and is controlled via the run_test() method. All of the vmm_xactor components, such as the stimulus generator (env.gen) and the driver (env.mst), are controlled by the phase methods of the vmm_env class.

## INTEGRATING OVM AND VMM

With the vmm_env class successfully instantiated in OVM, additional OVM functionality can be added to enhance the original testbench. Consider:

```
1) module example_02_add_ovm_sb;
2)     class env extends ovm_env;
3)         tb_env env_vmm;       // VMM env
4)         ovm_apb_sb #(ovm_apb_rw) sb;  // OVM component
5)         function new (string name="env",
                      ovm_component parent=null);
6)             super.new(name,parent);
7)         endfunction
8)         virtual function void build();
9)             env_vmm = new();
10)            sb = new("sb",this); //could use the factory
11)         endfunction
12)         virtual function void connect();
13)            ovm_apb_master_cb cb;
14)            cb = new(sb.analysis_export);
15)            env_vmm.mst.append_callback(cb);
16)         endfunction
17)         virtual function void end_of_elaboration();
18)            env_vmm.gen.stop_after_n_insts = 5;
19)         endfunction
20)     endclass
21)  initial
22)   run_test("env");
23)  endmodule
```

Along with the vmm_env class, we instantiate an OVM scoreboard component in the build method. Then, in the OVM connect() method, we create a VMM callback object (line 12–13) that converts the vmm_data transaction into an ovm_transaction and writes it to the

---

Figure 1: The default phasing order
scoreboard. This callback is then appended to the VMM's master driver (line 15), which was created as part of the build process.

In the end_of_elaboration() method, we can similarly configure the VMM generator (line 18). This is important because, unlike OVM, VMM components can only be configured after they have been built. Note also that the vmm_env class' gen_cfg() method will have been called prior to its build() method being called, as part of the OVM phasing mechanism.

The data conversion is accomplished via a user-defined convert() method inside the callback object. An example of the conversion from a vmm_data type (apb_rw) to a corresponding ovm_transaction type (ovm_apb_rw) is shown here:

```plaintext
function ovm_apb_rw convert(apb_rw from,
    ovm_apb_rw to=null);
    if (to == null)
      convert = new;
    else
      convert = to;
    case (from.cmd)
      apb_rw::READ: convert.cmd = ovm_apb_rw::RD;
      apb_rw::WRITE: convert.cmd = ovm_apb_rw::WR;
    endcase
    convert.addr = from.addr;
    convert.data = from.data;
    convert.set_transaction_id(from.data_id);
    convert.set_sequence_id(from.scenario_id);
endfunction
```

Note that the convert method will copy and convert the "from" object into the "to" object, if it is provided; otherwise it will create a new "to" object. The important concept to understand is that it is up to the user to write the conversion function, which can then be used throughout the system wherever such conversion is needed. There may also be an analogous convert method to convert from OVM back to VMM.

Additional OVM stimulus may be added and connected to the VMM driver via its vmm_channel pointer. Because the communication semantics of the vmm_channel are slightly different than for OVM TLM-based components, the Interoperability library also includes a set of adapters that present the OVM interface on one side and the VMM interface on the other. The adapter set allows OVM components, including sequencers and drivers, to be connected to VMM channels and to VMM atomic or scenario generators (via the inject() and inject_obj() methods, respectively). The full set of adapters can be seen by downloading the library.

ENCAPSULATING LEGACY VMM COMPONENTS IN OVM

The requirements specified by the VIP-TSC deal primarily with the interconnected model, where the integrator must be aware of both the OVM and VMM use-models in order to make them work together. In the previous example, the integrator is actually the test writer, since all VMM details are visible at the “top level” of the testbench. However, part of the reuse advantage of the OVM comes from its built-in notion of hierarchy, in which any component, including ovm_env and ovm_test, may be instantiatted by any other component. The flexible architecture of OVM ensures that the phasing is handled automatically throughout the hierarchy.

With this in mind, the question of incorporating legacy VMM VIP into an OVM-based testbench becomes even more straightforward, allowing the VMM legacy components to be wrapped in an OVM component and thus appearing to the test writer as if the testbench is completely OVM. All that is needed is to take the VMM-specific pieces of the system shown above and incorporate them in a wrapper component, which is extended from the same ovm_vmm_env proxy component discussed above. Consider:

1) module example_07_ovm_atomic_gen;
2)   class ovm_apb_env extends
    ovm_vmm_env_wrapper_noname #(tb_env);
3)     ovm_blocking_put_export #(ovm_apb_tr) inject_export;
4)     ovm_analysis_port #(ovm_apb_rw) ap;
5)     int num_trans = 100;
6)     apb_atomic_gen_adapter atomic_gen;
7)   function new (string name="ovm_apb_env",
     ovm_component parent=null);
8)       super.new(name,parent);
9)       atomic_gen_port = new("inject_export",this);
10)      endfunction
11)     virtual function void build();
12)       super.build(); // builds underlying vmm_env
13)     atomic_gen = new("vmm_atomic_gen_adapter",this,env.gen);
14)    get_config_int("num_trans",num_trans);
15)   env.gen.stop_after_n_insts = num_trans;
16)     endfunction
17)   virtual function void connect();
18)     ovm_apb_master_convert_cb cb;
19)   endfunction
```
The ovm_vmm_env_wrapper_noname component is an extension of ovm_vmm_env and includes the virtual methods for controlling the phasing of the vmm_env, which is specified as a parameter (line 2). This wrapper class includes much of the same functionality discussed in the previous interconnected example, with a few important differences.

First, because the wrapper component is now the parent of the vmm_env class it instantiates, we have a bit more flexibility regarding when to call particular phasing methods. Upon completion of super.build() on line 12, we are guaranteed that the underlying vmm_env class is built, so the configuration setting previously done in end_of_elaboration can now be done in the wrapper’s build() method directly. To the test writer, the configuration parameter is now set using the set_config_int() method (line 39), as it would be for any OVM component. The wrapper calls get_config_int() as part of its build() (line 14) and then uses that information to reach down into the VMM environment to set the parameter, according to the VMM guidelines.

The wrapper also provides appropriate ports and exports to allow other OVM components in the top-level environment to connect to it as needed. In this case, there is a stimulus injector export (line 3), which provides an interface (line 23) for an OVM stimulus generator to drive additional stimulus into the vmm_env (line 39). It also shows an analysis_port (line 4) to which the converter callback writes transactions (lines 21-22), allowing additional analysis components, such as the scoreboard (line 29), to be connected (line 40) as to any OVM component.

CONCLUSION

Using the wrapper approach it is now possible to take any vmm_env class — indeed, any vmm_xactor — and wrap it in an OVM component to hide any VMM-specific details about the underlying VIP. Using the interoperability library, it is only necessary for the wrapper writer to understand the VMM enough to instantiate the proper converter objects, do the necessary data conversions and configuration calls, and make the appropriate ports and/or exports visible in the wrapper. Once that is done, the rest of the team can reuse the VMM VIP as if it were simply an OVM component, according to all of the OVM guidelines.

The interoperability library presented here thus provides a low-risk migration path from your legacy VMM VIP to a fully OVM-based environment. Once the VMM is wrapped, the flexibility of the OVM makes it a simple exercise to replace the underlying VMM with an OVM implementation (either ported or acquired from some other source) of the same functionality. Rather than having to start over from scratch, the user can manage the risk of moving from legacy VMM environments to take advantage of all the superior benefits of the OVM.

1 All enhancements to the VMM library are transparent to the user.

2 VMM environments (vmm_env) and/or transactors (vmm_xactor, vmm_subenv) may be reused via this approach. Unless otherwise specified, the term “component” will be used to refer to “a piece of VIP,” regardless of what specific VMM class is used to implement the VIP.
INTRODUCTION

This is the third and final article of a series aimed at helping you get started with OVM in a simple, practical way. The emphasis of the series has been on the steps you need to take to write working code.

In the previous two articles we explored the overall structure of an OVM class-based verification environment and saw how to assemble OVM verification components, run tests, and reconfiguring the verification environment. In this final article we explore sequences and sequencers, which are the main tools used to generate structured test stimulus.

The previous article was published just as version 2.0 of the OVM class library was being released. One of the most important innovations in OVM 2.0 was the revision and improvement of the sequence classes. This article describes the new, improved sequences available in OVM 2.0.

The code you see in this article will run with OVM-2.0, and can be downloaded from the Doulos website at www.doulos.com/knowhow.

A VERY SIMPLE SEQUENCER

The role of the sequencer in OVM is to generate a stream of transactions that can be fed into a downstream component, which could itself be another sequencer or a driver. You may remember from the previous articles that a driver is a verification component that converts an abstract transaction into a series of pin wiggles on a design under test (or DUT). The stimulus generated by sequencers can range from a simple stream of random transactions to a highly structured sequence representing a set of nested protocols.

Here is the transaction class that appeared in previous articles, updated for use with the OVM 2.0 sequencer class:

```verbatim
class my_transaction extends ovm_sequence_item;
    rand int addr;
    rand int data;
    rand bit r0w1;

    function new (string name = "",
                  ovm_sequencer_base sequencer = null,
                  ovm_sequence_base  parent_seq = null);
        super.new(name);
    endfunction: new

    constraint c_addr { addr >= 0; addr < 256; }
    constraint c_data { data >= 0; data < 256; }

    `ovm_object_utils_begin(my_transaction)
    `ovm_field_int(addr, OVM_ALL_ON + OVM_DEC)
    `ovm_field_int(data, OVM_ALL_ON + OVM_DEC)
    `ovm_field_int(r0w1, OVM_ALL_ON + OVM_BIN)
    `ovm_object_utils_end

dendclass: my_transaction
```
It is important that the user-defined transaction class extends \texttt{ovm\_sequence\_item}: this allows the transaction to be generated by a sequencer. It is also important to use the correct constructor arguments, which are different from those of other OVM classes. We will now give a line-by-line description of the code for a very simple sequencer so that you can start using sequencers right away:

```verilog
class my_sequencer extends ovm\_sequencer #(my\_transaction);

A sequencer is a class derived from \texttt{ovm\_sequencer} and specialized to generate transactions of a particular type, which in this example is \texttt{my\_transaction}.

```verilog`
\texttt{ovm\_sequencer\_utils(my\_sequencer)}
``` 

Every sequencer class should use the \texttt{ovm\_sequencer\_utils} macro. This provides a number of facilities, including the ability to generate the sequencer from the OVM factory. Note that the macro \texttt{ovm\_sequencer\_utils} is used to register the sequencer, rather than \texttt{ovm\_component\_utils} that we have seen used to register other components for automation.

```verilog
function new (string name = "", ovm\_component parent = null);
super\_new(name, parent);
```

The constructor for an \texttt{ovm\_sequencer} takes the same arguments as that of an \texttt{ovm\_driver}, namely a component instance name and a reference to the parent component. Note that the constructor must call the constructor of the base class or superclass.

```verilog`
\texttt{ovm\_update\_sequence\_lib\_and\_item(my\_transaction)}
endfunction : new
endclass : my\_sequencer
```

Each sequencer has an associated sequence library, which holds all of the sequences which can be generated by that particular sequencer. The macro \texttt{ovm\_update\_sequence\_lib\_and\_item} populates this sequence library with an initial set of sequences, based on the type of the transaction \texttt{my\_transaction}. The sequence library is actually an array of sequences from which sequences are selected to be run one-at-a-time.

The macro \texttt{ovm\_update\_sequence\_lib\_and\_item}, adds three standard sequences to the sequence library. These are:

- \texttt{ovm\_simple\_sequence}, which consists of a single item of the given transaction type.
- \texttt{ovm\_random\_sequence}, which selects and executes a series of sequences from the array, but excluding \texttt{ovm\_exhaustive\_sequence} and \texttt{ovm\_random\_sequence} itself. By default, this only leaves \texttt{ovm\_simple\_sequence}. The number of sequences is determined by the count property of the \texttt{ovm\_sequencer} class, and defaults to a random number between 1 and 10.
- \texttt{ovm\_exhaustive\_sequence}, which selects and executes every sequence in the array once, but excluding \texttt{ovm\_random\_sequence} and \texttt{ovm\_exhaustive\_sequence} itself. Again, this only leaves \texttt{ovm\_simple\_sequence} unless further sequences have been added.

Hence \texttt{my\_sequencer} is created with a library containing the above three sequences. When simulation starts, the sequence \texttt{ovm\_random\_sequence} will be selected and run by default. This sequence will choose a random number between 1 and 10 of other sequences to be selected and run. (You can change the upper limit to be something other than 10 by setting the property \texttt{max\_random\_count} of the sequencer.) The only sequence available to be run is \texttt{ovm\_simple\_sequence}, which itself generates a single item of type \texttt{my\_transaction}.

As things stand, our example would generate a sequence of between 1 and 10 randomized transactions, with the actual number being determined at random. We can choose to take control of the exact number of transaction to be generated by setting the count property of the sequencer. A good place to do this would be from a test:

```verilog
class my\_test extends ovm\_test;
...
virtual function void build;
super\_build();
set\_config\_int(".*\_m\_sequencer\", "count", 10);
...
```

The \texttt{set\_config\_int} call sets the count property of the sequencer to control precisely how many transactions are generated by the default sequence \texttt{ovm\_random\_sequence}. The sequencer is chosen using a string that represents the OVM hierarchical name of the component and can contain wildcards.
THE DRIVER

The transactions generated by a sequencer get passed either to another sequencer or to a driver. A driver is a component that injects the stimulus represented by the transaction into the design-under-test by wiggling pins. A user-defined driver class follows the same pattern as other OVM components, so we do not need to explain it line-by-line. Here is the complete source code for a simple driver component suitable for connection to my_sequencer.

```verbatim
class my_driver extends ovm_driver #(my_transaction);
  virtual dut_if m_dut_if;

  'ovm_component-utils (my_driver)

  function new(string name, ovm_component parent);
    super.new(name, parent);
  endfunction: new

  function void build;
    super.build();
  endfunction : build

  virtual task run;
    repeat(10)
      begin
        my_transaction tx;
        #27
        // seq_item_port is built into ovm_driver:
        seq_item_port.get_next_item(tx);
        // Wiggle pins of DUT
        m_dut_if.addr1 = tx.addr;
        // Tell sequencer we are done
        seq_item_port.item_done();
      end
  endtask: run

endclass: my_driver
```

The `m_dut_if` is a virtual interface for connecting the driver to the design-under-test, as described in the previous articles in this series. The `ovm_component-utils` macro and the `new`, `build` and `run` functions and tasks are standard ingredients in any OVM component. The interesting part of the `my_driver` component is the contents of the `run` task, because this is the part of the driver that fetches transactions from the sequencer.

The driver communicates with the sequencer using the `seq_item_port`, which is a pre-defined port specifically for making a connection to an upstream `ovm_sequencer`. This part of the code is a bit tricky, because the port is not defined in the `my_driver` class itself but is defined in the base class (or superclass) `ovm_driver` and is inherited by `my_driver`. It appears as if `seq_item_port` is being used without being declared, because the declaration itself is in the source code of the OVM class library.

The driver calls two methods of the `seq_item_port`: `get_next_item` to pull the next transaction from the sequencer, and `item_done` to tell the sequencer that it has finished with the transaction.

SEQUENCES

The `ovm_sequencer` is not restricted to generating single transactions. The next step is to introduce a user-defined sequence, and add it to the sequence library of the sequencer. Whereas a sequencer is a component, a sequence relates a set of data items (transactions and other sequences) and includes code to generate them in the proper order. A user-defined sequence extends the class `ovm_sequence`:

```verbatim
class my_sequence_1 extends ovm_sequence #(my_transaction);

  `ovm_sequence-utils(my_sequence_1, my_sequencer)

  my_transaction seq_item;

  function new(string name="", ovm_sequencer sequencer=null, ovm_sequence parent_seq=null);
    super.new(name, sequencer, parent_seq);
  endfunction: new

  virtual task body;
    `ovm_do(seq_item)
    `ovm_do_with(seq_item, {addr == 1;})
    `ovm_do_with(seq_item, {addr == 2;})
    `ovm_do_with(seq_item, {addr == 3;})
  endtask: body

endclass: my_sequence_1
```

The macro `ovm_sequence-utils` registers the sequence for factory automation and adds the sequence to the library of the given sequencer. At this point, `my_sequence_1` becomes one of the sequences that may get selected by sequence `ovm_random`.
sequence of my_sequencer. Alternatively, my_sequence_1 may be selected manually using one of several possible methods, described below.

The main behaviour of a sequence is defined by its body method. A sequence typically generates a series of sequence items, where each item may be either a transaction or another nested sequence. The macro `ovm_do` provides a simple way in which a sequence can generate a sequence item. `ovm_do` creates a new object (whose type is determined by the type of the seq_item variable), waits for whatever is downstream of the sequencer to be ready to consume the item, randomizes the item, then sends the item downstream. The variant `ovm_do_with` macro is passed an in-line constraint (using SystemVerilog constraint syntax) which is used when randomizing the item. The point about this mechanism is that the constrained randomization is applied at the last possible moment, just before the sequence item is consumed by the driver.

In this example, my_sequence_1 generates a sequence of four transactions, where transactions 2, 3, and 4 have addresses 1, 2, and 3 respectively.

Each new sequence is added to the sequence library of some sequencer; any sequences already existing in the sequence library of that sequencer are still available for selection. If we run the above code as is, ten sequences will be selected and run at random by my_sequencer, where each sequence is either ovm_simple_sequence (a single transaction) or my_sequence_1 (four transactions).

STARTING SEQUENCES MANUALLY

Having a sequencer run through all the sequences in its library at random fits well with the principles of coverage-driven verification. But sometimes you will want to start a particular sequence explicitly. This can be achieved in several ways.

Another more flexible way to start sequences is using the sequence selection and execution methods. Sequence execution can be controlled from within a user-defined sequence class or from the test:

```plaintext
class my_test extends ovm_test;
...
virtual function void build;
    set_config_string("*.m_sequencer", "default_sequence", "my_sequence_1");
...

task run;
    ovm_sequence_base seq;
    repeat(2)
        begin
            seq = m_env.m_sequencer.get_sequence( m_env.m_sequencer.get_seq_kind("my_sequence_1") );
            assert(seq.randomize());
            seq.start ( m_env.m_sequencer );
        end
...
```

The run method above selects the sequence named “my_sequence_1” from the sequence list of the sequencer, randomizes the sequence object, and then executes the chosen sequence on the sequencer. The method get_seq_kind looks up a sequence by name and returns the index number of the sequence in the sequencer’s list. The method get_sequence creates a new sequence object of the given kind. The sequence object is then randomized before calling its start method, which begins execution of the sequence on the specified sequencer.

Also, note that the count field of the sequencer is set to 0. Without this, the ovm_random_sequence in the sequencer’s library would still run in its own right, and sequences from the random sequence and my_sequence_1 would get interleaved. This can be a source of endless confusion to beginners learning to use OVM!

CONNECTING THE PARTS TOGETHER

Having seen the definitions of the sequencer and driver classes, let us now complete the picture by showing how they can be connected together. This is straightforward and similar to the code seen in previous articles. The only complication is knowing exactly which ports need to be connected.
As before, the `ovm_component_utils macro and the new, build and connect functions are standard ingredients of any OVM component. The build function instantiates the sequencer and the driver components, and the connect function connects together their ports.

The build function, instead of calling new to instantiate the components, calls the OVM create method. Although not being used to full advantage in this simple example, this is a sophisticated programming trick that allows considerable flexibility in choosing which versions of the sequencer and driver components actually get instantiated at run time. The create method is a factory method, and can use override information defined elsewhere in the OVM verification environment to select the actual type of the component being instantiated. A full discussion of factory methods is outside the scope of this article, but is described more fully in the online version.

The connect function connects the ports. This is where it is useful to know what you are doing, because neither seq_item_port of the driver nor seq_item_export of the sequencer are defined in the user’s code. seq_item_port is inherited from ovm_driver and seq_item_export inherited from ovm_sequencer. Once these two built-in ports have been connected, any transactions from the sequencer will be sent to the driver.

CONCLUSIONS

In this article we have seen simple examples of using the ovm_sequence and ovm_sequencer classes, but this is just the beginning. These two classes can be used to build nested, hierarchical and layered sequences in many forms, sufficient to model typical layered protocol stacks.

In this series of articles we have examined the rudiments of building an OVM verification environment from a practical viewpoint. You can find the source code for these examples together with online tutorials and further hints and tips at www.doulos.com/knowhow, and you can download the OVM class library itself from www.ovmworld.org.
The first eight years of the 21st century have seen leading companies in our industry adopting new methodologies for verifying their designs. These companies publish papers highlighting considerable benefits gained by implementing advanced verification technologies such as functional requirements tracking, Coverage Driven Verification (CDV), Assertion Based Verification (ABV), formal verification, constrained random stimulus, and most recently, the Open Verification Methodology (OVM). And yet, a significant number of customers continue to use HDL based testbench methods developed in the 1990’s or earlier. Why is this?

The most common reason for keeping to traditional verification practices is the lack of an obvious migration path to allow risk free adoption of new approaches. In this article we describe a useful approach which has provided a migration path for many customers who are beginning to adopt advanced verification techniques. The solution presented is the combination of Questa’s Multilanguage support, and the Verilog bind mechanism.

UNDERSTANDING HOW BIND WORKS

We have seen that SystemVerilog provides advanced verification capabilities. Now we show the role of bind in a multi-language environment.

Adding SystemVerilog code into an existing Verilog or VHDL DUT consists of three separate issues, for clarity it is important to keep these separate –

- Firstly, mixed languages. This is a capability of Questa allowing a component written in one language to be instantiated in a structural design written in another language. You cannot mix SystemVerilog language constructs (such as covergroups) directly into VHDL code, because that would violate the VHDL syntax rules. You can only instantiate a design unit into another, for example, a SystemVerilog module into a VHDL structural design where the VHDL was expecting to find an entity/architecture instantiated.

- The role of bind is simply to add an instantiation of code to an existing design unit (anywhere in the hierarchy) without touching the original code. It does this by appending the extra code to an existing design unit during elaboration. The end result is the same as instantiating a SystemVerilog module directly into the source file. Bind itself does not provide any multi-language capabilities, but can be used with the multi-language Questa capabilities described above to instantiate a SystemVerilog design unit into a VHDL or Verilog hierarchical design.

Are You in a “bind” with Advanced Verification?
by Virginie Mallez, Mark Peryer, Andy Walton, and Paul Williams, Verification Practice, Mentor Consulting (Europe)
Design code should be kept separate from verification code. The principle being that a means of measurement should not influence the thing that is being measured. It is possible to add SystemVerilog verification constructs such as covergroups directly into Verilog RTL code, but the result will be unsynthesiseable.

Using a combination of bind and multi-language support we can insert verification capabilities such as covergroups and assertions into existing HDL code. We can even add pre-prepared verification components (such as an AMBA MVC verification IP or assertion based monitors from the Questa Verification Library (QVL)) to any point in our Verilog or VHDL design without touching the original source code.

We will now describe how this capability is used, starting with the simple case of using bind in a pure Verilog environment, and then adding the complexity of multi-language.

THE MECHANICS OF USING BIND WITH VERILOG

_bind_ indicates that a design unit (Verilog module, interface, or program block) is to be instantiated immediately before the end of a specified module or interface. Given a SystemVerilog module containing assertions:

```verilog
module my_assertions(input logic sig);
  property ...
endmodule : my_assertions
```

Instead of writing:

```verilog
module mod;
  logic my_var;
  initial ...

  my_assertions uassert (.sig(my_var));
endmodule : mod
```

_bind_ allows the following:

```verilog
module mod;
  logic my_var;
  initial ...

endmodule : mod
bind mod
my_assertions uassert (.sig(my_var));
```

The "bind" statement can be in any compilation-unit available during elaboration. For example, the bind statement, the target module and the module to be bound can all be in separate files.

Note that bind does not add any functionality to the SystemVerilog language; it simply allows a more convenient way to insert assertion code into an existing module without editing the source file. Also note that it is not restricted to binding modules of assertions. Any code can be bound, and modules, interfaces and program blocks can all be inserted using bind, subject to the rules that would apply were the blocks instantiated directly.

There are two forms of bind, the first, is to bind to all instances of a modules so that the new code will be added to every instance. The second is to bind to a specific instance, where code is added only to that instance. The syntax for this is as follows:

```verilog
bind t1.dut.u1 my_assertions uassert (.sig(my_var));
```

It is also possible to bind to multiple instances of a specified module (mod) using the following syntax:

```verilog
bind mod:t1.dut.u1, t1.dut.u2, t1.dut.u3
  my_assertions uassert (.sig(my_var));
```

THE MECHANICS OF USING BIND WITH VHDL

To bind the same SystemVerilog assertions module into an existing VHDL design the following syntax would be used to bind to all instances of the entity called ent:

```vhdl
bind ent my_assertions uassert (my_var);
```

Or, more specifically, to bind to all instances of an entity-architecture pair (note that you cannot bind to a configuration) -

```vhdl
bind 'ent(a) my_assertions uass2 (my_var);
```
Note that the escape token (\) used before the ent(a) is required in the bind statement above to allow the Verilog compiler to parse the VHDL architecture reference. Also note that no spaces are allowed in this identifier e.g. “\myent ( \myarch )” is illegal.

In exactly the same way as shown in Verilog, it is possible to bind to a specific instance of a VHDL design unit as follows:

```
bind tst.dut.u1 my_assertions uass3 (my_var);
```

There is no direct equivalent of the verilog multiple instance bind command, therefore binding to multiple (but not all) instances of a component would be written as:

```
bind tst.dut.u1 my_assertions uass1 (my_var);
bind tst.dut.u2 my_assertions uass2 (my_var);
bind tst.dut.u3 my_assertions uass3 (my_var);
```

It is also possible to bind to generated instances and an entity in a library, but these are less common - see the Questa documentation for details.

The SystemVerilog wrapper that makes the bind is either instantiated within the hierarchy of a SystemVerilog testbench, or the wrapper can be added as a top level at the simulator command line as follows:

```
vsim design sva_binding
```

**BIND ACROSS LEVELS OF HIERARCHY**

We have seen how the combination of bind and Questa multi-language support gives the ability to add advanced SystemVerilog modules to any component or instance in an HDL design. However, when binding modules into VHDL designs we are restricted to only access signals that are local to that specific component, or instance of that component. There are often cases where we need to monitor signals across multiple levels of VHDL hierarchy; for example, coverage of the interaction of two parts of the design, or an assertion that is deactivated by a reset applied at a parent level of hierarchy or sampled on a clock in another part of the design.

As an example, the following SystemVerilog code fragment counts the number of clock cycles where a state machine is in the wait state.

```
covergroup wait_cover @(posedge clk);
  wait_count:  coverpoint wait_count iff (encode_enable == 1) {
    bins counter_zero = {0};
    bins counter_one  = {1};
    bins counter_ok   = {[2:7]};
    bins counter_over = {[8:1023]};
    bins counter_problem  = {[1024:$]};
  }
endgroup
```

Logically the module containing this covergroup would be bound with the RTL code of the state machine, but the control signal ‘encode_enable’ is not available inside the hierarchical level where we have added the new covergroup. The solution is to bring all signals of interest up to the top of the design and process them there. We do this with the same bind mechanism to grab the interesting signals using a grabber/ripper block as shown here:

```
module sva_binding
  bind ent my_asserts sva1 (.a(a), .b(b), .c(c));
endmodule
```

In the diagram to the right SystemVerilog binding has been used to insert ‘signal grabbers’ into the VHDL design. These grabbers are SystemVerilog modules and as such are able to write hierarchical references (see next section) between the sub-design and a top level SV Interface. Coverage is then performed at the top level where all required information has been made available by the grabbers.
REFERENCES TO INSTANCES CREATED USING BIND

It is legal to use references to and into bound components. A common example of this is to group all signals of interest in a sublevel of the DUT into a SystemVerilog Interface. This Interface can then be referenced hierarchically from a top level testbench. For example group all signals on an AHB bus together into a SystemVerilog Interface using bind, and then refer in the testbench to a covergroup in that interface (e.g. dut.alu_sub_axi_bind.data_coverage) or to a task in that interface (dut.alu_sub_axi_bind.writeData(errorInjectedByte)).

BIND RESTRICTIONS

Since bind is the equivalent to inserting code into a source module, all restrictions that would apply when instantiating a component still apply. For example:

- The instance name must be unique in the target module. Thus, it is not possible to bind a component using an instance name that is already used in the target module. Also, multiple bind statements to the same module must not use the same instance name as each other. (VHDL & Verilog language rule)
- A component cannot be bound into a program block. (SV rule)
- Only an interface can be bound into an interface. (SV rule)

In addition, there are a number of bind-specific restrictions:

- Since the binding only occurs at elaboration time, there is no visibility of compilation-unit scope ($unit) to the bound module.

If the source file was edited and the instantiation added instead of being bound, the $unit namespace would be visible.
- The order in which bind statements are elaborated is not defined, and consequently it is not legal to rely on bind order.
- bind components cannot contain further bind statements.

USING BIND TO TAKE ADVANTAGE OF ADVANCED VERIFICATION

Bind can be used to bind SystemVerilog assertions and covergroups to RTL code without having to make any changes to the design code.

This means that development teams can easily start to use assertions alongside their traditional testbench, gradually building from simple debug assertions to sophisticated protocol monitors as their fluency improves. It also means that they can take advantage of covergroups to get functional coverage information from their traditional test benches.

Our experience is that once the power of these two methodologies is grasped, and then there is a natural progression towards wanting to improve stimulus generation using advanced techniques such as constrained random.

SUMMARY

bind is a simple and effective way to incorporate verification code into existing RTL designs. The bind statement contains a specification of where a component is to be bound – i.e. all instances of a component; specific instances of a component; or a specific instance; and the instantiation that is to be incorporated at the specified location in the hierarchy:

\[
\text{bind } \langle \text{where} \rangle \text{ }\langle \text{instantiation-as-though-added-by-editing}\rangle;
\]

Remembering this simple structure should help avoid any confusion in the use and operation of the bind statement.

Use of bind and Questa’s multi-language support provides a mechanism for adding SystemVerilog constructs to an existing HDL design / testbench and is therefore a valuable part of a low risk migration path to advanced verification.