Hi everyone, and welcome to another super-sized Verification Horizons issue for DVCon U.S.

Last spring, my wife and I decided to take a family vacation to Hawaii this February, which coincidentally is just before DVCon. We’re celebrating several family milestones, including our 20th anniversary, my son’s 18th birthday and my father-in-law’s 80th, so we’re all going. As I’ve mentioned before, my wife is great at planning and managing things like this, and there are many similarities between planning a vacation and managing a verification project. As we know, having a plan is critical to a project’s success, so my endearingly “old-school” wife has all of our plans written out in a document, which she’ll print out and bring with us. As a “tool guy,” I showed her an app I use that will automatically load all our confirmation emails into a cool interactive itinerary. It’s great to have a plan, but automation and tools are what make the plan into something usable. You’ll see this theme throughout the following articles.

We begin this issue with two case study articles from users “in the trenches” of verification. First, our friends at Baker Hughes share “An Evaluation of the Advantages of Moving from a VHDL to a UVM Testbench,” in which they discover the advantages of self-checking randomized testing in UVM, even for FPGA designs. For those of you doing FPGA designs in VHDL, this article should allay any fears you may have about moving to UVM as most of your competitors are doing.

Our second case study comes from our friends at Qualcomm, with an assist from XtremeEDA, where they share their “First Time Unit Testing Experience Report with SVUnit.” Their methodology stresses unit testing critical testbench components to avoid the dreaded “is it a design bug or a testbench bug?” question that so often plagues verification engineers, particularly at the integration stage. As you’ll see, this approach does require some up-front effort, but the payoff is clear. If you can prevent bugs from getting through to tapeout, why wouldn’t you?
We begin a set of articles from my Mentor colleagues by introducing “The Verification Academy Patterns Library,” a new feature of the Verification Academy website that documents a set of good design practices to solve often-recurring problems in verification. The concept of design patterns is not new, but we believe this is the first and most extensive effort to document a pattern library specifically for verification. As you’ll see, the pattern library is clearly organized into categories so it will be easy to locate a pattern that may be applicable to your specific problem and allow you to take advantage of the knowledge provided by a diverse team of experts from assertion-based and formal verification to constrained-random and coverage-driven verification across simulation, hardware-assisted verification and emulation.

Next we learn how to achieve “Increased Efficiency with Questa® VRM and Jenkins Continuous Integration” by applying the software practice of Continuous Integration to verification management. Experience and common sense show that the longer a branch of code is checked out the more it drifts away from the previous version in the repository, making it more likely that problems will occur when checking it back in. The article shows how Jenkins, a free open-source tool, can be used to monitor the source repository and use Questa’s Verification Run Manager (VRM) to handle the necessary verification tasks and supply results back to Jenkins for display in a dashboard.

Our next several articles highlight different aspects of Questa Verification IP (QVIP), beginning with “Verifying Display Standards: A Comprehensive UVM-Based Verification IP Solution.” This article offers practical advice on how to set up your UVM environment to include QVIP as well as highlighting some of the benefits of QVIP in general. In “Nine Effective Features of NVMe® Questa® Verification IP to Help You Verify PCIe-Based SSD Storage,” you’ll get an overview of the new Non-Volatile Memory Express® (NVMe) specification and see how our new Questa NVMe VIP can help you accelerate the verification of your PCIe-based Solid State Drives that use the NVMe interface.

In “MIPI C-PHY™: Man of the Hour,” you’ll get an introduction to the three physical layers used in the MIPI Alliance for mobile imaging systems and the tradeoffs between them, and learn what features Questa VIP provides to assist in their verification. We wrap up the QVIP articles with “Total Recall: What to Look for in a Memory Model Library,” which provides an extremely useful analysis of the key features you should look for in evaluating a VIP Memory Library. It highlights some of the unique features of the QVIP Memory Library, including on-the-fly configuration.

Our next article, “Certus™ Silicon Debug: Don’t Prototype Without It,” addresses that age-old question of what to do once you’ve gotten your full SoC running as an FPGA prototype in the lab and you find a problem. It highlights the many layers of the debug problem and shows how our Certus™ Silicon Debug tool provides unsurpassed visibility into the inner workings of your FPGAs and lets you see the results in the Visualizer™ Debug Environment, just as if you were running in simulation. The idea of defining trigger conditions and capturing HW signals reminds me of my days designing logic analyzers back in the 80s (yes, I’m that old), and I find it fascinating that we can now do the same thing inside an FPGA with millions of gates. This is some really great technology that you have got to check out.

Next we have the first of several articles relating to DO-254 verification. We begin with “Simplified UVM for FPGA Reliability,” where we see how the component-based nature of UVM can help with the auditing process in DO-254. This article also reiterates some of the conclusions from the Baker Hughes article.
In our Partners’ Corner, we continue our DO-254 sub-theme with a discussion of “Complex Signal Processing Verification Under DO-254 Constraints,” in which our friends at AEDVICES Consulting show how they combined assertions, UVM and functional coverage to support requirements-based verification for safety critical processes like DO-254 and ISO 26262.

Since no DO-254 project is complete without documentation, our friends at eInfochips walk us through “Simplifying Generation of DO-254 Compliant Verification Documents for Airborne Electronic Hardware (AEH) Devices.” They show us a step-by-step process to go from a Verification Case Document (VCD) to importing a testplan into a UCDB in Questa, against which you can measure your functional coverage from your UVM simulation. We follow this with a discussion of “DO-254 Compliant UVM VIP Development” from Electra IC, in which they provide a case study of putting together a UVM environment using Questa VIP and Verification Run Manager for a recently completed DO-254 project. And last but not least, we learn from our friends at Ensilica how to build a “Reusable Verification Framework,” where they use UVM to build BFMs in the interface instead of virtual interfaces in the driver to simplify block-to-top reuse of interface components.

I hope to see you at DVCon. I’ll be around in many of the sessions, speaking on a few panels, and you can always stop by booth 501 to say “Hi”. I’m hoping to show off my tan.

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons
Table of Contents February 2016 Issue

Page 6
An Evaluation of the Advantages of Moving from a VHDL to a UVM Testbench
by Shaela Rahman, Baker Hughes

Page 12
First Time Unit Testing Experience Report with SVUnit
by Neil Johnson, XtremeEDA Corp.
Nathan Albaugh, Qualcomm Technologies, Inc.
Jim Sullivan, Qualcomm Technologies, Inc.

Page 16
The Verification Academy Patterns Library
by Harry Foster, Michael Horn, Bob Oden, Pradeep Salla, and Hans van der Schoot, Mentor Graphics

Page 20
Increased Efficiency with Questa® VRM and Jenkins Continuous Integration
by Thomas Ellis, Mentor Graphics

Page 26
Verifying Display Standards
A Comprehensive UVM-based Verification IP Solution
by Saumya Agrawal and Manish Chand, Mentor Graphics

Page 32
Nine Effective Features of NVMe® Questa® Verification IP to Help You Verify PCIe® Based SSD Storage
by Saurabh Sharma, Mentor Graphics

Page 39
MIPI C-PHY™: Man of the Hour
by Yogesh Chaudhary and Vikas Sharma, Mentor Graphics
Page 43
Total Recall: What to Look for in a Memory Model Library
by Mark Peryer, Mentor Graphics

Page 46
Certus™ Silicon Debug: Don’t Prototype Without It
by Doug Amos, Mentor Graphics

Page 50
Simplified UVM for FPGA Reliability
UVM for “Sufficient Elemental Analysis” in DO-254 Flows
by Shashi Bhutada, Mentor Graphics

Partners’ Corner

Page 58
Complex Signal Processing Verification under DO-254 Constraints.
by François Cerisier, AEDVICES Consulting

Page 64
Simplifying Generation of DO-254 Compliant Verification Documents for AEH Devices
by Hari Patel and Amarkumar Solanki, eInfochips

Page 68
DO-254 Compliant UVM VIP Development
by Ates Berna, ELECTRA IC

Page 71
Reusable Verification Framework
by Gunther Clasen, Ensilica
FPGA designs are becoming too large to verify by visually checking waveforms, as the functionality has become beyond easy comprehension. At Baker Hughes, a top-tier oilfield service company, we primarily design small scale FPGA designs, typically less than 100 thousand gates, but our designs are growing in size and complexity. As well, they are part of more complex systems that require long lab integration times.

For these reasons, our FPGA design and verification team needed to find a new methodology that does not rely on visual checks, shortens our lab integration testing time, standardizes our testbenches, and makes our functional verification more robust.

VHDL was the language of choice for both our RTL design and testbench, but there was no standardization or robustness for our testbenches. We did not reuse our testbenches, though we had the intention to do so. We knew the Universal Verification Methodology (UVM) provided testbench standardization, which is why we wanted to investigate the adoption of a UVM-based flow.

In order to justify the transition from a VHDL to a UVM testbench, we conducted an evaluation to compare the pros and cons of using three styles of testbench: a simple VHDL testbench, an advanced VHDL testbench, and a UVM testbench. The metrics used to evaluate them were the level of code coverage achieved, the reusability of testbench components, and the ease of creating different stimuli.

PREVIOUS DESIGN AND VERIFICATION FLOW
Most of our designs are debugged during lab integration testing. Our design and verification flow begins with an RTL design and a testbench created in VHDL. We apply simple stimuli to our design under test (DUT) and visually inspect the waveforms using a simulator. Once we achieve some confidence in our visual inspection, we go into the lab and download the code to an FPGA and integrate it into the rest of the design, which includes several PCBs. If we see the FPGA is not working correctly during the integration, we modify the testbench and check the waveform again by visually inspecting it. We repeat this loop until no bugs are found, which takes several weeks or months.

Most of our designs, which go into oil and gas extraction tools, have to withstand high temperature and vibration conditions. So when we pick a component, such as an FPGA, we put it through a series of qualification processes at operating conditions. Once the FPGA is qualified, we do board-level qualification, and then we do system-level qualification until we are confident the final tool is ready to be deployed in the field.

It is important to note that most of our FPGA failures are found during system-level testing. So there are a lot of other components that we are qualifying along with the FPGA. The entire system-level test can take up to a year or two, so if we can significantly reduce the time and increase the thoroughness of the FPGA debug and verification effort, it will positively impact our overall product development schedule.

THE EVALUATION DESIGN AND TESTBENCHES
For this evaluation, we used a small-scale FPGA design that has three interfaces: a GPIO, an ADC, and a UART interface. The GPIO and the ADC are the inputs to the DUT, and the UART is the output. Thus, the testbench stimulates the GPIO and ADC interfaces and reads from the UART interface.

![Figure 1: The evaluation design block diagram.](image-url)
For the evaluation, we first created a simple VHDL testbench, then transitioned that into a VHDL advanced testbench, and finally transitioned that to a UVM testbench.

Through this transition process, all three testbenches remained equivalent in terms of what they model. Whatever was modeled in the simple VHDL testbench was ported over to the advanced VHDL testbench and then to the UVM testbench. Even though the level of code coverage achieved is one of the metrics, no specific changes were made for the sole purpose of increasing code coverage as that would have compromised the commensurability of the testbench comparisons.

The simple VHDL testbench used RTL-style processes. The advanced VHDL testbench used records for the GPIO, ADC, and UART transactions and used procedures for stimulus and checking. In the UVM testbench, components called “agents” were used for the GPIO, ADC, and UART interfaces. A key aspect of the UVM environment we built was that the test sequences for stimulus generation were isolated. Now let’s look at each in more detail.

**TESTBENCH 1: A SIMPLE VHDL TESTBENCH**

In this testbench, the DUT, stimulus, and checks were modeled in RTL-style processes. The checks were modeled mostly after debugging in the lab. We created a simple set of tests and checked the waveforms visually. There were some simple assertions included after we inspected the waveforms visually, but a more comprehensive testbench was not achieved until after lengthy lab testing.

This is a highly reactive process for verifying a DUT. Every time you find a problem during the lab integration testing, you have to go back to the testbench, modify it, recreate the bug that caused the error, fix the design, and then go back to the lab to test it again.

The block diagram of the simple VHDL testbench shows the DUT with the three interfaces. There are multiple processes for each of the interfaces. There are four processes for the GPIO, three for the ADC, and two for the UART. The checks are scattered here and there. This reflects a lack of organization because all these processes and checks are added every time more bugs are found during lab integration testing. This makes it difficult for even the person who created the testbench to follow what is going on.

What we learned during the evaluation was that the simple VHDL testbench requires the least amount of time up front to create the testbench, but it is very disorganized and hard to follow. Also the focus is on creating stimulus, not checks. What we want is to concentrate on creating both.

**TESTBENCH 2: AN ADVANCED VHDL TESTBENCH**

We did not want to jump directly from the simple VHDL testbench to the UVM testbench without checking out advanced VHDL constructs. We knew that UVM used transaction-level modeling, so we tried to create that in the advanced VHDL testbench using records and procedures. Each DUT interface is connected to a record, which is analogous to a SystemVerilog interface. Then we added procedures that either stimulated or monitored the records. The procedures were then encapsulated in a process.
designated to call only that procedure. The self-checking portion (what’s called the scoreboard in UVM) is modeled in a separate checker procedure. There were three processes to generate the stimulus and monitor the bus activity and one process to validate the results.

With only four processes, the advanced VHDL testbench has fewer processes than the simple VHDL testbench. Each stimulus process sends whatever it generates to a CHECK process. This evaluates whether the UART packets equaled what the GPIO or the ADC process stimulus generated and is equivalent to a scoreboard. Thus all the checks are in one process, giving this testbench much better organization. The more organized structure made it easier to follow than the simple style testbench, and also a lot easier to understand when you go back and read the code.

Figure 3: Advanced VHDL testbench diagram.

Figure 4 records equivalent to transactions (‘adcRecType’ is an ADC transaction and “pktRecType” is a transaction sent to the check process, or scoreboard).

Figure 5 shows procedures equivalent to a driver, monitor, and scoreboard (procedure “storePacket” is the scoreboard that checks ADC transaction data).

Figure 6 shows top level processes that call the procedures for each interface. Processes pass around records (or transactions).

What we learned was that there is definitely better code organization than with the simple VHDL testbench, but changing stimulus for different test sequences is still difficult because you need to re-write the procedures and make significant changes to the drivers. The procedures and records can be put in packages for reuse, but there is no standardized methodology to follow. The testbench is still monolithic. For all of these reasons, if somebody else wanted to use this testbench, connecting it to their DUT would not be intuitive.

TESTBENCH 3: UVM

The UVM testbench was created using the UVM library, with SystemVerilog as the testbench language. The interfaces to the DUT were partitioned into several different classes; i.e., driver, monitor, and agent. An agent was created for each interface. An agent is a container for stimulus and verification components for an interface. Thus, agents are important for code reuse.

The testbench environment contains all the agents to the DUT and the scoreboard. The scoreboard is designed to self-check results. A top-level test class contains the testbench environment and the test sequences (See Figure 7).

Again, we have the DUT in the center and the three interfaces, but now agents are connected to the interfaces. The GPIO and ADC agent each have a driver and a monitor. The UART agent has only a monitor because the DUT is only transmitting on the UART interface. Thus, while the GPIO is driving the DUT, it’s listening at the same time via the monitor. The monitor then captures and sends that information to the scoreboard.

![UVM Diagram](image)

Figure 4: Equivalent to transactions.
In the UVM testbench, information about the interfaces at the system level is passed around using transactions. These transactions are generated by the test sequences. The test sequences come up with different test stimuli for the GPIO and ADC transactions. The transactions are generated and passed into the agents, and then passed into the DUT. A copy of that gets passed into the scoreboard.

The scoreboard does automatic checking and is always checking whether the UART packets are equal to the ADC and GPIO transactions.

Critically, the test sequences are isolated from the agents and the testbench environment. This is very important because when you change the test sequences, you don’t have to change any of the agent’s drivers.

In comparison, with the VHDL testbenches, you have to do significant changes to the drivers, procedures, and processes, which you won’t do unless you find bugs in the lab. It takes too long to do otherwise.

This is a big advantage of the
UVM testbench. Because the test sequences are isolated in UVM, you can create another test very quickly in order to improve your testing. In this case, we were able to go from 79% to 85% code coverage by changing the test sequence alone — from ordered to randomized triggers.

Changing a test sequence and re-running code coverage in the UVM environment takes less than a day. In comparison, with a VHDL style testbench it would take several weeks to achieve a similar improvement.

The lessons learned with the UVM testbench are that the test sequences are isolated from the structural testbench and that gives better control in that you can change the stimulus without redesigning the agents. Code coverage was increased easily by changing the test sequence. Using UVM supports an industry standard, so it will encourage testbench standardization. For VHDL there is no testbench industry standard. UVM also improves reusability through object oriented programming (OOP) features, such as inheritance, and by using override components, which are allowed by polymorphism. These are additional benefits of UVM.

**SUMMARY AND RESULTS**

Comparing the testbenches in light of our metrics can be summarized as follows.

VHDL does not offer much support for constrained random stimulus. SystemVerilog does; thus in the UVM testbench the ADC and GPIO can be randomized with constraints. Very little of the simple VHDL testbench can be reused. The advanced VHDL testbench can be reused by creating packages, but connecting to a new user testbench is not intuitive because there is no standardized methodology. Furthermore, because packages are structural they organize only code. They cannot handle the explosion of additional functions required every time you extend a record into a new data type. Plus, they split the data definition from the function definition (i.e., the package body); whereas, OOP ties the two together for easier maintenance. The UVM testbench handles all of these things elegantly because of the OOP features, resulting in highly reusable verification components and infrastructures.

Also with both VHDL testbenches, engineers are compelled to focus more on creating stimulus — not checks. What we want is to concentrate on creating both the stimulus and checks. And that's what you get in UVM.

Code coverage achieved for the VHDL simple testbench was 78%. It was 74% for the VHDL advanced testbench. Recall that the aim in moving from the simple to the advanced VHDL was not to improve code coverage per se. The intent was to use advanced constructs (i.e., records and procedures) and have a more organized testbench. Code coverage for the UVM was 79% percent when using the first test sequence, where the triggers were in order, and 85% with the second test sequence, where the triggers were randomized. The ease of making this improvement is what is important here.

**Figure 8: Comparison of testbenches.**

<table>
<thead>
<tr>
<th></th>
<th>VHDL Simple</th>
<th>VHDL Advanced</th>
<th>UVM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Method</strong></td>
<td>RTL style using processes</td>
<td>Records for GPIO, ADC, UART transactions. Procedures for stimulus and checking</td>
<td>UVM agents for GPIO, ADC, and UART interfaces. Built environment and isolated sequences for stimulus generation</td>
</tr>
<tr>
<td><strong>Randomized</strong></td>
<td></td>
<td><strong>Stimulus</strong></td>
<td>Yes, ADC and GPIO randomized with constraints</td>
</tr>
<tr>
<td><strong>Reusable</strong></td>
<td>No</td>
<td>Yes, by creating packages. Connecting to user testbench is not intuitive</td>
<td>Yes. OOP features allow reuse</td>
</tr>
<tr>
<td><strong>Code Coverage</strong></td>
<td>78.59%</td>
<td>74.41%</td>
<td>79.71% with 1st test</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>85.58% with 2nd test</td>
</tr>
</tbody>
</table>
The UVM testbench increases robustness of functional verification, and allows you to create flexible, adaptable, scalable testbenches. It offers quantifiable, measurable progress, and changing test sequences can be used to easily increase code coverage. UVM reporting, which we did not discuss, is very powerful and was used in our reviews.

Ultimately, we believe that UVM will reduce our tool integration time by allowing us to attain our ultimate goal of decreasing the time of our lab integration cycles.

Finally, UVM takes away the focus from doing visual waveform checking, and that’s important because FPGAs are becoming too big to check the waveforms visually. Therefore, eventually we are going to have to use UVM for the verification of FPGAs.
INTRODUCTION
Verification teams don’t typically verify testbench components. But this Qualcomm Technologies IP team realized the necessity of unit testing a critical testbench component and the corresponding debug time and frustration it could prevent for downstream IP and chip teams.

This experience report documents the team’s first time unit testing with SVUnit, from start to finish. We discuss the justification made with managers to have engineers assigned, how to pick the right test subject to ensure a positive outcome, the method used for verifying testbench checkers, component defect rate, framing unit testing and the defects found as opportunities for improvement and lessons learned for long term maintenance. The focus of the test method is the SVUnit UVM report mocking to write automated tests for uvm_error checks.

WHY IS DEBUG ACCEPTABLE IN SOC DEVELOPMENT?
Debug has become a fundamental part of SoC development. So fundamental, in fact, that in 2014 verification engineers estimated 37% of their time was spent debugging code. Assuming an eight hour day, 37% is roughly two hours and 53 minutes a day, everyday, spent debugging issues that were more than likely created by the development team itself. For hardware developers this is a horrid statistic that signals an obvious breakdown in development.

Data from the same survey suggests that adoption of advanced verification methods is on the rise. While the proliferation of advanced verification methods may be perceived in industry as a positive sign, there’s been no noticeable change in mean time lost on debug for the six year period dating back to 2010. This suggests that advanced verification methods have failed to help engineers produce the high quality code necessary to avoid debug. Time lost to debug, therefore, continues with no probable end in sight.

BREAKING THE DEBUG CYCLE
From a quality perspective, the data suggests that not only are the methods we use to produce code inadequate, the advanced methods being encouraged and adopted within industry provide little in the way of improvement. To break the trend, therefore, teams should consider that:

- Poor initial code quality is leading to irresponsible bug rates and substantial time and money lost
- Focusing on and/or relying on advanced verification techniques and industry best practices to rectify the loss is a flawed strategy
- A more effective solution is likely to come from new industry practices

These are the three steps this team went through on their way to unit testing a critical UVM testbench component with SVUnit.

THE UNIT UNDER TEST (UUT)
The ABC is a critical component of the XYZ SoC. It is a bridge/interconnect that facilitates a high bandwidth connection between large processor subsystems and various memory and peripheral subsystems. There are multiple instances of ABC on the XYZ SoC which magnifies its importance.

Initial versions of the ABC were verified using a typical approach. A block level ABC testbench was created to verify it in isolation. Once verified in isolation, the ABC was integrated and verified in larger subsystems and finally the XYZ SoC testbenches. The block level ABC testbench was a self-checking constrained random testbench supplemented by embedded SVAs.

The typical approach to verifying the ABC produced typical issues. Namely, bugs undetected in the block level verification went on to require debug investments in dependent subsystem and XYZ SoC tests.

Because of the importance of the ABC and corresponding bug rate, the group manager overseeing testbench
development initiated a discussion around improving code quality and eliminating bugs. Having heard of SVUnit and being familiar with unit testing, he suggested a unit testing pilot project focused on one problematic component within the ABC testbench: the DEF checker.

For a number of reasons, the DEF checker was an excellent starting point for improving the quality of the ABC testbench because:

- Many of the bugs found in the ABC testbench were attributed to protocol violations that should have been flagged by the DEF checker
- Protocol violations intended to be flagged by the DEF checker were well documented in a series of tables
- While protocol rules were believed to be fully implemented in the DEF checker, there was low confidence in the correctness of each rule
- Loose coupling between the DEF checker and surrounding testbench meant the DEF checker could be easily isolated and tested with few external dependencies
- The DEF checker flagged protocol violations using uvm_errors which could be captured and validated using the SVUnit UVM report mock

Following an hour long presentation to leadership responsible for the XYZ SoC, it was decided that three weeks would be dedicated to the unit testing pilot in hope unit testing would avoid further time lost to debug of the ABC.

THE SVUNIT UVM REPORT MOCK

Unit testing of the DEF checker relied heavily on SVUnit and the SVUnit UVM Report Mock. The SVUnit UVM report mock enables automated testing of uvm_errors to increase confidence that testbench checkers are defect free. It is a scoreboard style checker where actual and expected errors are logged and compared to decide a PASS/FAIL result.

Unit tests typically go through the following steps to verify a uvm_error is being flagged properly:

- Set the expectation of a particular error by calling the svunit_uvm_report_mock::expect_error(...) function
- Apply a violation to the UUT
- Call svunit_uvm_report_mock::verify_complete() to ensure actual errors are flagged by the UUT as expected. (The verify_complete() function returns 1 when expected and actual errors match; 0 otherwise.)
- Terminate the test with an SVUnit assertion based on the return value of svunit_uvm_report_mock::verify_complete()  

To illustrate, we refer to code snippets from the SVUnit UVM Report Mock example packaged with SVUnit. In the first snippet, we have a UUT with a method called verify_arg_is_not_99(). In this trivial example, verify_arg_is_not_99() is designed to flag a uvm_error for input arguments set to 99; all other values are ignored.

```c
function void verify_arg_is_not_99(bit [7:0] arg);
  if (arg == 99) 'uvm_error("uut", "arg is 99!");
endfunction
```

To test the verify_arg_is_not_99() function, we can write a set of SVUnit unit tests. This first test validates the error condition of 99 is properly flagged by a uvm_error.

```c
// desc: when 99 is passed in, there should be an errors flagged. We expect that error by first calling the uvm_report_mock::expect_error()
// expect:
'SVTEST(_99_is_an_error)
  uvm_report_mock::expect_error();
  my_uut.verify_arg_is_not_99(99);
  'FAIL IF(uvm_report_mock::verify_complete());
'SVTEST_END(_99_is_an_error)
```

The test _99_is_an_error follows the bullet steps outlined previously. One uvm_error is expected thus the expect_error() method is called once. Calling the expect_error() method pushes an item onto the expected queue of the log message scoreboard inside the SVUnit UVM Report Mock.
The UUT function is then called with an argument of 99. Assuming correct behaviour, this should result in one item being pushed onto the actual queue of the log message scoreboard. Finally, we terminate the verify_complete in a FAIL_IF assertion to do an in-order comparison between the actual and expected queues thereby confirming a match between expected and actual errors.

The mechanism not shown in the tests is a redirection of uvm_errors to the SVUnit UVM Report Mock instead of the normal UVM reporting facilities. This is done by redefining the uvm_error macro to call the svunit_uvm_report_mock::actual_error(...) function. The macros are redefined as follows:

```
define uvm_error(ID,MSG) \ uvm_report_mock::actual_error(MSG, ID);
define uvm_fatal(ID,MSG) \ uvm_report_mock::actual_fatal(MSG, ID);
```

To exhaustively verify a checker, tests can also be written for happy path scenarios to ensure uvm_errors are not erroneously flagged. For example, we may wish to verify that values other than 99 do not flag a uvm_error from the verify_arg_is_not_99() function. The following test, other_numbers_are_not_an_error, does just that. No errors are expected, the function is called for all possible arg values other than 99 and the verify_complete() is called to ensure the expected number of errors - in this case 0 - are flagged by the UUT.

```
// test: other_numbers_are_not_an_error  // desc: no other 8-bit numbers should cause an error
// SVTEST(other_numbers_are_not_an_error) for (int i=0; i<255; i+=1) begin
  if (i != 99) my_uut.verify_arg_is_not_99(i);
end

FAIL_IF(!uvm_report_mock::verify_complete()); SVTEST_END(99 has a specific message)
```

With expected MSG and ID arguments, the verify_arg_is_not_99() function must trigger the uvm_error with the expected MSG and ID; simply flagging a uvm_error is not enough. (NOTE: for the DEF checker unit tests, no MSG or ID arguments were specified. As long as uvm_errors were flagged as expected, the message content of the uvm_errors was deemed unimportant.)

Using SVUnit, the expect_error() function, the redefined macros and the verify_complete() function, checking of uvm_errors can be automated in unit tests. The automated tests replace crude eyeball checking of error logs and/or blind trust that checkers behave as intended.

**PROCEDURE AND RESULTS FROM THE DEF CHECKER**

To test the DEF checker, 109 unit tests were written to validate a total of 65 protocol rules. Most protocol checks required exactly one unit test while others required two or more to cover obvious corner cases. The SVUnit UVM Report Mock was used to automate checking as described in the previous section.

Only basic scenarios were used to validate uvm_errors fired for each protocol violation. While the DEF checker did include checks for more complex scenarios, those scenarios were not tested due to time constraints.

In dedicating one verification engineer to write focused unit tests over a period of 13.5 days, a total of 12 bugs were found in the DEF checker. In some cases, bugs were found in the implementation of a check. In others, the intended protocol check was missing entirely.

Of all 12 bugs found in the DEF checker, two were masking bugs in the corresponding ABC RTL implementation. Therefore, as a result of unit testing the DEF checker, two bugs were prevented from infiltrating real XYZ SoC silicon.
These results matched the expectation of the verification team; that bugs remained in the DEF checker even after more than a year of intense use in both subsystem and SoC tests. They also marked the end of a successful unit testing pilot.

LESSONS LEARNED
In this unit testing pilot, a number of lessons were learned:

- Buy-in from managers and the foresight to allocate three weeks to unit test the DEF checker was key. Without it, the development team may or may not have captured the bugs found and defects may or may not have made it to silicon.
- Sensible code partitioning and loose coupling between verification components made writing focused unit tests straightforward. The ease at which the DEF checker could be decoupled from the surrounding testbench made it very easy to directly isolate, stimulate and validate protocol checks.
- Code dependencies between the DEF checker and surrounding testbench were broken wherever possible in the interest of short simulation runtimes. Depending breaking and the reduced code base lead to runtimes of roughly two minutes 30 seconds for the entire unit test suite running on a single CPU.
- Reinroducing real dependencies at the conclusion of the unit testing effort would have protected against code changes breaking the unit test suite. This was not done though until follow-up testing easily broke the unit test suite through simple changes to the DEF checker and its dependencies. Runtime with all dependencies in tact was four minutes 30 seconds for the entire unit test suite running on a single CPU.

SUMMARY
Debug is only inevitable because development teams use process and techniques that result in poor initial code quality. The time and money we lose to debug is something we make possible. Through recognition of this fact and by changing development habits we can reduce the time and money wasted on debug.

While industry does not yet consider unit testing an advanced verification technique, unit testing nevertheless proves to be a very straightforward and methodical technique for improving code quality. For this team, allocating three weeks of effort to unit testing the DEF checker was a productive way to eliminate bugs. Unit testing not only improved the quality of the DEF checker, it increased confidence in the ABC testbench and prevented two bugs from being taped-out on the XYZ SoC; all for relatively minimal effort.
INTRODUCTION

The literature for many of today’s testbench verification methodologies (such as UVM) often reference various software or object-oriented related patterns in their discussions. For example, the UVM Cookbook (available out on the Verification Academy) references the observe pattern when discussing the Analysis Port. One problem with the discussion of patterns in existing publications is that it is generally difficult to search, reference, and leverage the solutions these patterns provide since these publications are distributed across multiple heterogeneous platforms and databases and documented using multiple varied formats. In addition, most of the published examples of a verification design deal more with the software implementation details of constructing a testbench. To address these concerns, we have decided to extend the application of patterns across the entire domain of verification (i.e., from specification to methodology to implementation—and across multiple verification engines such as formal, simulation, and emulation) and have just released a comprehensive pattern library out on the Verification Academy.

But first, we should answer the question, “What is a pattern?” In the process of designing something (e.g., a building, a software program, or an airplane) the designer often makes numerous decisions about how to solve specific problems. It would be nice if the knowledge gained from solving a specific problem could be shared, and this is where patterns help out. That is, if the designer can identify common factors contributing to the derived solution in such a way that it can be applied to other similar recurring problems, then the resulting generalized problem-solution pair is known as a pattern. Documenting patterns provides a method of describing good design practices within a field of expertise and enables designers to improve the quality in their own designs by reusing a proven solution on a recurring problem.

Design patterns are not a new concept. In fact, they originated as a contemporary architectural concept from Christopher Alexander in 1977, and they have been applied to the design of buildings and urban planning[1]. In 1987, Kent Beck and Ward Cunningham proposed the idea of applying patterns to programming[2]. However, it was Gamma et al., also known as the Gang of Four (GoF) who popularized the concept of patterns in computer science after publishing their book Design Patterns: Elements of Reusable Object-Oriented Software in 1994[3].

ORGANIZING THE PATTERNS IN OUR LIBRARY

Our Verification Academy Pattern Library contains a collection of pattern entries—where each documented pattern entry provides a solution to a single problem. To facilitate learning, ease of use, and quick access when searching for verification pattern content, we gave careful thought into organizing the library into searchable categories whose patterns solutions are related and exhibit similar characteristics. Since our goal in creating verification patterns is to broaden the application of patterns beyond the software domain, we decided that our categories should align from a high level with the digital design and verification process. Hence, we have identified two main verification pattern categories, which should be familiar to any design and verification engineer working in this domain. That is, Specification Patterns and Implementation Patterns, as illustrated in the following figure.

DOCUMENTING PATTERNS

We believe that it is important, when creating a pattern library, that each documented pattern follow a consistent format and style. This consistency simplifies learning and facilitates ease of use when reviewing different patterns contained in the library. The documentation for a verification pattern should describe the context in which the pattern is used—a problem within this context that the pattern is seeking to address—and a suggested solution.
All patterns within the Verification Academy Patterns Library adhere to the following template, which can be downloaded out on the Verification Academy so that users can document their own patterns:

- **Pattern Name:** A unique (descriptive) name that helps in identifying and referencing the pattern.
- **Intent:** A very brief description of the goal behind the pattern and the reason for using it.
- **Motivation:** A description of a specific scenario consisting of a problem and a specific context in which this pattern can be applied. Think of this as a problem statement that describes a concrete example (the solution to the problem will be discussed in the subsequent Implementation and Example sections).
- **Applicability:** Situations in which this pattern is usable; the general context for the pattern.
- **Structure:** (Required for Implementation Patterns and optional for Specification Patterns) Abstract graphical representation of the pattern (e.g., UML class diagrams, interaction diagrams, etc.).
- **Implementation:** A description of an implementation of the pattern; the solution part of the pattern.
- **Example:** A code (or pseudo-code) example of how the pattern can be used. It is suggested that the example addresses the original problem scenario presented in the motivation section.
- **Scope:** (Recommended for Specification Patterns and optional for Implementation Patterns) A scope defines the extent of the verification execution over which the pattern must hold. More property scope details will be discussed in section 3.2, and an example is provided in section 4.1.
- **Consequences:** (Optional) A description of the results, side effects, and tradeoffs caused by using this pattern.
- **Related Patterns:** (Optional) Other patterns that have some relationship with the pattern with a discussion of the differences and similarities between the related patterns.
- **Contribution:** Identification of person and/or references for this pattern contribution to the library.

**PATTERN EXAMPLE**

The following example is provided to help illustrate how we document a pattern in the Verification Academy Pattern Library using the template we just described in the previous section. This unique example demonstrates a pattern that is targeted at both simulation and emulation.

**Pattern Name:** The BFM-Proxy Pair Pattern.

**Intent:** The BFM-Proxy Pair Pattern is categorized as an Environment Pattern and facilitates the design of transactors like drivers and monitors for dual domain partitioned testbenches that can be used for both simulation and emulation, and across verification engines (or platforms) in general.

**Motivation:** In order to enable and promote a verification process that is abstracted from underlying verification engines, particularly a software simulator and a hardware emulator, modern testbenches should exhibit (from conception) a dual domain architecture with partitioned HVL and HDL module hierarchies targeted for the simulator and emulator, respectively, and linked together to run in unison. Fundamental to this architecture is the employment of BFM-proxy pairs to devise so-called split transactors, where components in the HVL domain typically implemented as classes act as proxies to BFM implementations as interfaces or modules in the (synthesizable) HDL domain. An HVL proxy provides a surrogate or placeholder for the associated cross-domain HDL BFM to control access to it via a transaction-based HVL-HDL communication model using remote function and task calls. Effectively, the proxy embodies the transactor API to upper testbench layers, abstracting the cross-domain communication and the implementation details of the BFM’s bus cycle state machines.

**Applicability:** The BFM-Proxy Pair Pattern is applicable in any situation demanding a common dual domain partitioned testbench architecture (i.e., separated HVL and HDL module hierarchies) for both simulation and emulation, and across verification engines in general.
Structure: The diagrams below illustrate the dual domain testbench architecture and the according UVM agent structure, respectively, with the transactors depicted as BFM-proxy pairs.

Example: BFM-Proxy Pair Pattern source code examples for a UVM driver and monitor are provided on the following page:

Consequences: The dual domain partitioned testbench architecture enabled by this BFM-Proxy Pair Pattern offers maximum leverage of established simulation-based verification practices into emulation, including the benefits of using SystemVerilog and UVM for creating modular, reusable verification components and environments.

Related Patterns: A precursor to this BFM-Proxy Pair Pattern is the Dual Domain Hierarchy Pattern, which advocates the HVL and HDL domain partitioning as a sound and necessary separation of concerns fundamental to emulation and other hardware-assisted verification platforms. Additionally, the BFM-Proxy Pair Pattern resembles the proxy pattern as one of the structural patterns of the GoF's OOP design patterns (though applying instead between a dynamic proxy object and a static interface or module).

Creating a Community of Pattern Expertise
For the Verification Academy Patterns Library, we felt it important to set goals on the pattern creation process and how to effectively populate the library. On a related note, you might have wondered why there is such a large set of authors listed on this article (we refer to ourselves as the Gang of Five). The reality is that verification is a diverse field, and it often requires expertise in varied areas, such as methodologies, technologies, tools, and languages. No single person is a master in every aspect of verification. Thus, to create patterns across the broad field of verification, we built a team made up from experts in assertion-based verification, formal verification, constrained-random and coverage-driven verification, UVM, hardware-assisted verification, and emulation. However, even with this diverse team of experts we recognize that there is still additional verification expertise required for solving verification problems in specific application domains. Hence, for our verification patterns library, we set a goal...
that the pattern creation process should harness the power of online social communities made up from a diverse set of verification experts that work in multiple application domains. In turn, this community of experts would foster collective problem solving for the creation of novel patterns and provide alternative, optimized solutions for existing pattern content. To achieve these goals, we developed a web-based infrastructure that allows new content to be contributed in a consistent format from this community of experts, and decided to release our library out on the Verification Academy, since it consists of an existing online social community with over 35,000 design and verification engineers. In addition, the Verification Academy provides us an existing online infrastructure, which enabled the creation of a patterns knowledge base that is easily discoverable, referenceable, and relatable.

To learn more about the Verification Academy Patterns Library, check out www.verificationacademy.com.

REFERENCES

“Time is really the only capital that any human being has, and the only thing he can’t afford to lose.”
—Thomas Edison

For all the incredible technological advances to date, no one has found a way to generate additional time. Consequently, there never seems to be enough of it. Since time cannot be created, it is utterly important to ensure that it is spent as wisely as possible. Applying automation to common tasks and identifying problems earlier are just two proven ways to best utilize time during the verification process. Continuous Integration (CI) is a software practice, which is focused on doing precisely that, resulting in a more efficient use of time.

WHAT IS CONTINUOUS INTEGRATION?
The basic principle behind Continuous Integration is that the longer a branch of code is checked out, the more it begins to drift away from what is stored in the repository. The more the two diverge, the more complicated it becomes to eventually merge in changes easily, ultimately leading to what is commonly referred to as “integration hell”. To avoid this, and ultimately save engineers time, CI calls for integrating regularly and often (typically daily).

Regular check-ins are of course, only half the equation; you need to be able to verify their changes quickly as well, otherwise many small check-ins over several days, is no different than one large check-in at weekends. Commonly, in a Continuous Integration environment, a CI server monitors the source control for check-ins, which in turn triggers a CI process (time-based triggers are also common). This process will then build the necessary design files, and run the requisite integration tests. Once complete, the results of the tests are reported back to the user, and assuming everything passed, can now be safely committed to the repository.

By following this model, issues can be caught earlier in the development process, and can be resolved quicker as there is less variance between check-ins.

This practice has been used successfully for many years in the software industry, so much so, that it is fairly common place today. However, the idea of Continuous Integration is still fairly new in the realm of hardware verification, so it is difficult to find any published metrics on its usage as it pertains to that space specifically. However, one of the benefits of adopting a more mature technology, is you can avoid making some of the pitfalls which plagued early adopters. Since Continuous Integration technology has been used by software teams for some time, you can glean a general idea of both how widespread its usage has become, as well as what technologies have risen to the top.

ZeroTurnaround is a development company, which amongst other things, conducts an annual global survey of Java developers, and produces a report of the tools and technologies being most commonly used by the industry. In 2014, they received responses from nearly 2200 developers covering many topics, one of which was their usage of Continuous Integration Technologies. In that survey, they found that roughly 80% of (or four out of five) developers, reported using Continuous Integration in their teams. A number which itself, showed fairly significant growth, up from 68% the prior year.

Another interesting aspect of the report, is the breakdown of which Continuous Integration servers were most commonly used. Far and away the most popular server was Jenkins, which was reportedly used by 70% of the developers who claimed to use CI. The second place tool was used by a mere 9% of users. So what is Jenkins, and why is it the favorite CI tool of so many users?
MEET JENKINS

Jenkins is a freely available, open-source continuous integration tool (released under the MIT license).

A quick background, Jenkins was initially developed by Kohsuke Kawaguchi while he was working at Sun Microsystems in 2004. However, at the time, the project was named Hudson. After its initial release in 2005, it quickly became a favorite open-source build server. In 2010, issues began to arise between the open source community working on Hudson, and Oracle (who had since acquired Sun). Eventually requiring a vote to be called, as to whether or not to fork the project. Based on an overwhelmingly supportive community vote, Jenkins was created as a fork of Hudson. The majority of those working on, or using Hudson at the time, eventually migrated to Jenkins. Currently there are at least 127,000 installations of Jenkins (based on the anonymous usage statistics of the tool). Remember the ZeroTurnaround study? They found only 8% of users to still be using Hudson.

Apart from being open-source, Jenkins is easy to install and highly configurable via its web interface. While Jenkins offers a lot itself, it is also highly extensible via plug-ins to the tool. At present, it boasts 1350+ plugins from 580+ contributors, to perform a myriad of different tasks, allowing for many third-party tools to leverage the power of Jenkins.

JENKINS AND VRM

On the surface, one might think that Jenkins and VRM are competitive technologies; after all, both tools can build, run and report on regressions. However, in actuality, they are truthfully complementary technologies. Furthermore, by marrying the two technologies together, you can benefit from the strengths of both, and create an extremely powerful solution for building and testing hardware designs.

While Jenkins is extremely flexible, and can run just about anything, with lots of neat bells and whistles to boot, nothing within the Jenkins core is knowledgeable about hardware verification. In the same way that VRM does not natively monitor code repositories for developer check-ins, concepts like merging SystemVerilog functional coverage, or recognizing why a UVM testbench failed are not native to Jenkins, in the way that they are at the core of VRM. What you want to do is leverage Jenkins’ strengths as a build system to monitor our source repository and allow it to launch our regressions. Ultimately what it will launch though, is VRM, which will handle managing the individual verification tasks by integrating with our grid software, collecting and merging the coverage and results, etc. Once the regression is completed, Jenkins can then ask VRM to supply metrics for what was accomplished during the run, and display those results in its web dashboard.

A REGRESSION IN JENKINS

Let’s take a quick look at setting up a project to run VRM in Jenkins. On the following page is the project configuration page in Jenkins.
Here you can see the basic steps for configuring a project in Jenkins. Tasks in Jenkins are represented by builds. A build could be a complete regression, it could be the running of unit tests, or any other task you may wish Jenkins to automate.

First you specify when to run our tests via a build trigger. A build trigger can be a period of time, a specific time, or you can even have Jenkins monitor your repository for changes, and automatically start a build for you. Jenkins will then run whatever you tell it to, which in this example, will be to launch VRM.

Finally, Jenkins will report the results of the regression run (or build). Out of the box, Jenkins will give you basic pass/fail information and some basic reporting of results, however, its lack of the metrics verification engineers are most commonly interested in, makes it feel a bit empty. To solve this shortcoming, you need one last piece to truly tie everything together neatly.

**VRM JENKINS PLUG-IN**

As mentioned earlier, one of the key benefits of Jenkins is that it is highly extensible through plug-ins. To get Jenkins to become more useful with respect to you running regressions with VRM, you can leverage the VRM Jenkins plug-in. You simply install the plug-in through Jenkins plug-in manager, and now Jenkins has the ability to understand code and functional coverage, determine where log files reside, monitor host utilization, and many other verification centric tasks.

To display the VRM results, and enable these features, you simply need to add what is called a post-build action (in Jenkins terms), which has Jenkins call the plug-in to make sense of the regression results.
The setup is very straightforward, you simply need to tell Jenkins where the regression ran. Additionally, you can optionally select to enable a few other features such as creating HTML reports and publishing a coverage graph to the project page. That’s it! Jenkins and VRM will do the rest.

**VRM REGRESSION RESULTS IN JENKINS**

One of the great features of Jenkins is its web dashboard. Now that it is using the VRM Jenkins Plug-in, you get access to a lot of great information at a glance. There is far too much to show in this short article, but here are a few examples.

The main project page has two graphs which shows you a trend of the test results, as well as the coverage results from all your past regression runs. You also get a summary table which lists the last several regressions, including information on their duration, pass/fail statistics and coverage. There are also quick links to the HTML coverage report, as well as the latest test results.

If you dig into the most recent build, you can get more detailed data on that particular regression.

---

**Figure 7: Jenkins Project Dashboard**
Here, in addition to pass/fail and coverage results, you can also see a list of the specific tests which failed, providing a means for easy high-level inspection. Expanding a given test, will give us both the reason for the failure, as well as the standard output for the test in question.

The plug-in leverages the vast amount of data VRM collects from the regression runs, allowing for all sorts of data to be analyzed that would otherwise need to be collected and reported manually. Otherwise difficult questions become easy to answer. Has this test, with this seed ever failed before? What is the host utilization like during a nightly regression? When did coverage drop off?

**Summary**

Continuous Integration with Jenkins CI, coupled with Questa Verification Run Manager, provides a powerful automated solution for build and regression management. By automating the regression process and helping to identify problem areas earlier, they allow verification engineers to make more efficient use of the time given even in the tightest of schedules.
Figure 10: Regression Data Examples
INTRODUCTION

The display protocol IP market is growing at a very fast pace. This is chiefly the outcome of the incredible increase in popularity of a wide variety of display source devices: such as DVD players, computer systems, and display sink/receiver devices: such as televisions, projectors, and display instruments. End users, the consumers, have also become more technologically savvy, increasing the demand for more and better products.

With this rapid growth in sophisticated display-dependent devices, there is an even faster rate of growth required in the field of verification of such devices. The verification of these devices must occur within very tight schedules, which demands a user-friendly solution so verification engineers can spend verification cycles in a productive and effective manner. This article provides a unique solution that is far superior to other solutions present in the industry. It allows engineers to design structured, reusable VIP that is easy-to-use and makes it easy to debug issues.

TECHNOLOGY OVERVIEW

The display IP segment involves transferring images and in some cases audio of various types and dimensions. Protocols like HDMI, Embedded DisplayPort, V-by-One, and others, as defined by multiple organizations, fall under this category. Along with these, there are various content protection mechanisms specified by various HDCP versions.

The basic function of any display specification is to send an image from a source or transmitter and receive it via a sink or receiver. Since all of the protocols have the purpose of sending images (a video is also an image sent at a high rate), there are inherent similarities in their nature.

The similarities of the specifications start from the fact that all of them have a period of active pixels (which are eventually bytes of data) during which pixels of the image are sent. Also, there is a period of blanking, which usually carries definite control and/or audio/secondary data (which are again bytes of data), as shown in Figure 1. Different specifications support different ranges of pixel format; for example, RGB and YCbCr.

VERIFICATION CHALLENGE

To understand the verification challenges involved and arrive at the best solution, we must answer two questions.

1. Why is it difficult to verify this kind of a display protocol?
2. Should there be a common solution for the verification of these industry standards?

The major challenges in the functional verification of display protocols are as follows:

- Initially a verification engineer may want to do a simple pipe-cleaning. Bring up takes a big part of the verification cycle due to the inherent complexity of the protocols.
- Due to the large frame size, generally a lot of data needs to be handled.
- Different specifications provide a variety of secondary data, with different flavors within each of them. In order to completely verify all of these, support for different combinations is required.
- There are various types of encoding/decoding and scrambling/descrambling and other techniques like encryption involved. This further makes the debug of issues difficult as the data on the interface is not the actual pixel/secondary data.
- Due to the large size of the frames (each targeting different features of pixel/secondary data), the pixel depth supported, and the combination of frame formats, it can be difficult to generate frame stimulus without easy-to-use APIs.

Verifying Display Standards
A Comprehensive UVM-based Verification IP Solution
by Saumya Agrawal and Manish Chand, Mentor Graphics
Considering these and other challenges confronting the fast growing display industry, it is inevitable that a good verification solution is needed. Such a solution must support reuse at a protocol-independent, higher-level for basic testing/pipe-cleaning and then address the intricacies of individual specifications as a separate step using a separate API. These intricate APIs verify features at a fine-tuned level and, hence, will vary from specification to specification.

When a design under test (DUT) is a source (or transmitter), it needs to correctly transmit the supported formats, making sure no protocol timing and compositions are violated. Correct and timely reading (or writing whenever required) of display capability memories is also essential. Further, all supported types must be verified thoroughly along with a check on the content of the audio/video data. Link set up/trainings steps should be complied as per the specification.

When a DUT is a sink (or receiver), it is necessary to receive all the supported types of frames/images and, side-by-side, correctly handle the ones not supported. Proper maintenance of display capability memories is also needed. Moreover, any link training or bring up as per the specification should be adequately supported as well.

This article goes through a solution provided by display family of Questa® Verification IP (QVIP) that targets all the issues specified above so that a verification engineer is able to verify the design effectively. The display family QVIP provides solution for verification of many display protocols which are industry standard, such as DisplayPort, HDMI, V-by-One.

**CREATING THE TESTBENCH**

The first and foremost requirement for a verification engineer is to set up a testbench that can verify their DUT. QVIP provides easy bring up components and APIs for this testbench. Connection modules are provided that take away the pain (and potential mistakes) of having to remember the directions of the signals when doing the signal assignment. A user simply selects the module that reflects the way they want the QVIP to behave. For example, if the QVIP is to be a source, select the source QVIP component. Then the engineer just passes the signals via the port list and the connections are handled by the QVIP itself. See Figure 2.

![Figure 2: Connecting the DUT with QVIP.](image)

The next step is to set up a UVM environment. The protocol specific agents make this simpler. QVIP provides protocol specific agents for each protocol, which is controlled by simple configuration switches. Refer to Figure 3 for details.

![Figure 3: Protocol specific QVIP agent.](image)

Various static time functionalities are easily controlled via the agent itself. For example, in order to configure the QVIP to behave as a display source, a simple setting of a configuration does the work. Further, examples are present with the QVIP to explain all use cases. Using the connection module and protocol specific agent, the basic testbench set up is performed with ease. Although, this is a generic step for any verification, it is still an important one,
as spending a lot of time here is not justified when there is a lot to be done elsewhere.

**STIMULUS GENERATION**

While verifying a sink or receiver DUT, one of the major efforts in verification has to do with writing the perfect stimulus to check all possible scenarios. The steps to generate the stimulus should be simple enough so that engineers can focus more on thinking of interesting scenarios rather than spending time and effort on how to code those. To verify a sink DUT, the engineer needs to transmit the frame/image through a source VIP.

As a first step of verification, a user may want to perform basic pipe-cleaning. This is basic verification that does not involve detailed protocol knowledge or specific pixel data values. Only a basic level of timing and dimension checks for the image kick-start the verification. A verification engineer does not want to spend a lot of time bringing up this level of stimulus generation. All that is needed is a very simple API, which sends some non-zero, random pixel data. QVIP targets this requirement very cleanly. It provides an easy to use API called send_frame.

The user follows a two-step process (see Figure 4):

1. Create a sequence extended from `<protocol>_source_seq`
2. In the body task of the sequence, call the API send_frame()

Starting this sequence on the agent’s sequencer sends a frame with random values in active pixels. Such a clean API provides a hassle free solution.

At this point, the user may want to check some timing with some audio (or secondary) data. This again requires some easy steps to fill the various attributes from a large variety of audio data and to place them at a particular frequency in the blanking period.

The APIs set_audio does the work here. It contains arguments as simple enumeration values for packet types (as per the specification) and other detailed fields. A user may want to set the packet types at the broad level, allowing QVIP to set the other detailed fields as per the specification (at the same time randomizing the ones which have multiple options). They can leave the arrays for the detailed attributes empty in such a case.

Alternatively, a user may want to fine tune the various attributes by themselves. If this is the case, they can set the required fields with a variety of set attribute APIs.

Thus, using the set_audio API, various combinations of audio (or secondary) data can be verified.

In the next step, the verification engineer needs to verify the pixel data by sending a particular value of pixel data.

- The pixel data may be directly available to the user. If so, the set_video API can be used directly.
- A png image of the intended frame may be available instead of direct pixel data values.

In the second case, the image is converted from an image format to a pixel format using an image convertor sequence. The converter sequence is started inside an image sequence (which is extended from the `<protocol>_source_seq`) and the pixel file is created. The pixel file is read and pixel components are extracted from the file.

Source sequence APIs are used to set the frame formats and send the frame. The image components are provided through the send_frame API to sequence the item. The BFM then does the necessary calculations to send the bytes to the correct wires of the interface connected to the DUT. The DUT (sink/receive, in this case) may extract the pixels and convert them to the image format for comparison purposes (the transmitted and received images can then be compared); see Figure 5.

```verilog
class my_seq extends hdmi_source_seq;
    `uvm_object_utils(my_seq)
    function new(string name="");
        super.new(name);
    endfunction

    virtual task body();
        super.body();
        send_frame();
    endtask

endclass
```

*Figure 4: Sending basic stimulus.*
Various other components of a frame (e.g., color depth and pixel format) can be set through the attributes API present in the source sequence.

**USING QVIP AS A RECEIVER/SINK**

The stimulus generation discussed in the previous section pertains to the use case when the DUT is a sink. In the other use model, where the DUT is the source, the challenges discussed earlier are handled very gracefully. The frame of an image can be easily received by the sink sequence and an actual image generated out of it. A comparison of this image can then be done with the image that the source DUT initiated. This kind of scoreboard makes it easier to verify the data to ensure that the DUT is functionally correct.

In some display specifications, a secondary channel is used to read/write display capabilities (DDC/EDDC, CEC) and requires HDCP operations that need to be responded to by the sink (or receiver). These are done by the QVIP along with various configurability options, which are present in order to vary the responses where the specification allows. Along with this, the user may vary signals, such as hot plug detect (or other sink end signals), which tells whether a sink is actually connected or not.

**COVERAGE COLLECTOR**

Once a user starts writing the test cases, the question arises: “when is it time to say that the verification is complete?” The coverage collector plays a critical role here. QVIP Coverage collector targets the specification extensively and makes sure that the design can be called completely verified if it achieves 100% coverage. The QVIP also provides an example that runs stimulus sequences to achieve 100% functional coverage and also a verification plan in the form of an XML file, which clearly maps individual sections of the respective protocol specifications. The Questa Verification Manager can be used for tracking the bins of the various coverpoints and crosses, as shown in Figure 6 and 7.

**Figure 6: Coverage window.**
CHECKERS AND ASSERTIONS
The first and foremost purpose of verification IP is to check for protocol compliance and give an error message when any protocol violation occurs. To fulfill this basic requirement, the QVIP contains assertions to check interface activity for protocol compliance. The assertions are structured with:

- The ability to enable/disable each assertion individually. By default, all assertions are enabled.
- Specific error messages that give a detailed description for the protocol violation along with the respective specification reference and make it easy to cross check and understand the reason for a violation.
- SystemVerilog Assertion (SVA) capabilities such as wave-level debug and assertion coverage.

A sample assertion firing is shown in Figure 8.

Although these features are a generic component to all QVIPs, in the display family QVIPs, the most important aspect becomes the debuggability of the assertions. This is because the large amount of data makes it practically impossible for a verification engineer to debug the cause of the issue. With intuitively structured error messages and high debuggability, the QVIP makes it very easy to understand the issues.

DEBUGGING ISSUES
As already discussed above, display protocols are inherently difficult to debug due to the huge amount of data. Over and above this, the data is not a direct reflection of its true value. The data is scrambled, encoded, encrypted in some cases, and then sent on the interface wires. By the time data reaches the interface, it’s completely different.

QVIP hides these intricacies of the functionality, but it does not compromise debuggability. Whenever something wrong happens on the bus, appropriate assertions are given along with the detail about the place of error. Then the sequence item at the point of error can be seen at various abstraction levels.

Even more, the performance and efficiency are very high, and frames of even very large dimensions can be easily executed on the interface at an efficient rate.

The large amount of data also makes it difficult to print out the active video, audio (or secondary) data, and users may want to observe various proceedings of a frame at a regular interval of time.

For this purpose, the QVIP provides a configurable control to print the timing of important events in the proceedings of the frame. For example, debug messages are seen when a vertical blanking completes, when each video line completes, when audio (or secondary) data packets are seen, and so on. Figure 9 shows a sample debug message.

---

Figure 7: Tracker window.

Figure 8: Assertion snapshot.

# ** Error: (vsim-60133) MVC @ 473500000000 fs: /top/dp_src/dp_if_aux.pkt: HDCP_BKSV_NON20_ZEROS_AND_ONES:155 - HDCP Receiver KSV should have 20 zeroes and 20 ones. See HDCP Spec 1.3 section 2.2.1 
# ** Note: (vsim-60000) MVC @ 473500000000 fs: /top/dp_src/dp_if_aux.pkt: Number of ones in ksv are 18
Figure 9: Debug messages.

Additionally, the basic feature of transaction viewing and transaction linking in the GUI makes the debug easier by many folds. This is again a feature common with all QVIPs but holds more importance in this context as it allows the user to find the issues in the sea of data that is available here. Lastly, the SVA features make it easier to debug the root cause of any protocol violation.

CONCLUSION

Considering all the above points, it is obvious that verifying a display component is not an easy task. But the Questa Verification IP provides easy-to-use, highly organized steps that make the verification experience more productive and ensures that the protocols have been implemented correctly, providing a high confidence in design quality.
INTRODUCTION

Non-Volatile Memory Express® (NVMe®) is a new software interface optimized for PCIe® Solid State Drives (SSD). It significantly improves both random and sequential performance by reducing latency, enabling high levels of parallelism, and streamlining the command set while providing support for security, end-to-end data protection, and other client and enterprise features. NVMe provides a standards-based approach, enabling broad ecosystem adoption and PCIe SSD interoperability.

This article provides an overview of the NVMe specification and examines some of its key features. We will discuss its pros and cons, compare it to conventional technologies, and point out key areas to focus on during its verification. In particular, we will describe how NVMe Questa® Verification IP (QVIP) effectively contributes and accelerates verification of PCIe-based SSDs that use NVMe interfaces.

To enable higher-speed client SSDs in the near term, the only choice was PCIe, which supports up to 1GBps per lane and has software-transparent multi-lane support for scalability (×2, ×4, ×8, and so on).

These limitations motivated development of a new software interface optimized for PCIe SSDs, NVMe.

TECHNOLOGY OVERVIEW

The NVMe specification by the NVM Express Workgroup is an optimized, high performance, scalable host.
controller interface with a streamlined register interface and command set designed for enterprise, datacenter, and client systems that use non-volatile memory (NVM) storage.

An NVMe controller is associated with a single PCI function. The capabilities and settings that apply to the entire controller are indicated in the controller capabilities (CAP) register and the identify controller data structure.

NVMe is based on a paired submission and completion queue mechanism. Commands are placed by host software into a submission queue. Completions are placed into the associated completion queue by the controller. Multiple submission queues may utilize the same completion queue. Submission and completion queues are allocated in memory.

An admin submission and the associated completion queue exist for the purpose of controller management and control (e.g., creation and deletion of I/O submission and completion queues, aborting commands, etc.). Only commands that are part of the admin command set may be submitted to the admin submission queue.

The interface provides optimized command submission and completion paths. It includes support for parallel operation by supporting up to 65,535 I/O queues with up to 64K outstanding commands per I/O queue.

Some of the key NVMe features are:
- Does not require un-cacheable / MMIO register reads in the command submission or completion path
- A maximum of one MMIO register write is necessary in the command submission path
- Support for up to 65,535 I/O queues, with each I/O queue supporting up to 64K outstanding commands
- Priority associated with each I/O queue with well-defined arbitration mechanism
- All information to complete a 4KB read request is included in the 64B command itself, ensuring efficient small I/O operation
- Efficient and streamlined command set
- Support for MSI/MSI-X and interrupt aggregation
- Support for multiple namespaces
- Efficient support for I/O virtualization architectures, like SR-IOV
- Robust error reporting and management capabilities
- Support for multi-path I/O and namespace sharing
- Developed by open industry consortium

The following figure explains the steps involved in NVMe command processing.
NVME QVIP FEATURES

1. Plug and play
In order to make the NVMe QVIP work out-of-the-box and easier to deploy, a number of integration kits are available for some commonly used PCIe design IP. These quick starter kits can be easily downloaded for serial or pipe link and are pre-tested with commonly used IP vendors. These kits are basically a reference resource to allow smooth integration of DUTs with UVM testbenches having QVIP and running first stimulus.

Also, single and multiple NVMe controller use-case examples are part of the standard deliverable database.

2. In-depth protocol feature support
NVMe QVIP provides extensive protocol feature support, allowing a wider degree of stimuli generation. Some of the supported features are:

- Complete admin and NVMe command set
- PRP and scatter gather lists
- Non-contiguous queue operations
- Metadata as extended LBA or separate buffer
- Namespace management
- End-to-end protection
- Security and reservations
- Host memory buffer
- Enhanced status reporting
- Controller memory buffer

3. Multiple controllers support
NVMe QVIP supports multiple controller operations. Since each PCIe physical or virtual, function, single root I/O virtualization (SR-IOV) can act as an NVMe controller, NVMe QVIP as a host can generate stimuli on the target controller or respond as a controller based on its location in the bus topology (bus, device, function number). In Figure 6 below is an example of multiple controllers attached to virtual functions of a physical function.

4. PI based stimuli
NVMe QVIP provides a rich set of APIs for easier stimuli generation. These APIs can be categorized as transactional, attribute/command specific, and utility based. They provide a simple interface to initiate traffic from a higher abstraction (transactional) or precise control at the command (command specific) or byte levels. Users can modify stimulus at the byte abstraction level for intentional error injection.

Figure 5: Quick starter kit scope

Figure 6: PCIe device supporting SR-IOV
5. UVM based register modeling

NVMe QVIP adopts UVM register modeling concepts and provides a way of tracking the register content of a DUT and a convenience layer for accessing register and memory locations within the DUT. In order to support the use of the UVM register package, QVIP has an adapter class, which is responsible for translating between the UVM register packages generic register sequence items and the PCIe bus specific sequence items (Figure 8).

A register model matching DUT properties can be built using register generator applications like Questa’s UVM register assistant or be handwritten in the csv format as shown in Figure 9 on the following page.

Register-based modeling allows users to write reusable sequences that access hardware registers and areas of memory. It is organized to reflect the DUT hierarchy and makes it easier to write abstract and reusable stimuli in terms of hardware blocks, memories, registers, and fields rather than working at a lower bit pattern level of abstraction. The model contains a number of access methods which sequences use to read and write registers.

The UVM package also contains a library of built-in test sequences which can be used to do most of the basic register and memory tests, such as checking register reset values and checking the register and memory data paths.
6. UVM reporting based protocol checker

To prove protocol compliance of the DUT, VIP must provide protocol checking. NVMe QVIP has built-in protocol checkers that continuously monitor ongoing bus traffic for protocol compliance and flashes a UVM message when there is undesired activity. These protocol checks cover the NVMe specification in depth and can be individually enabled or disabled at run time.

7. Easy to read text-based logs

NVMe related protocol traffic is captured in real time from the PCIe bus and printed in a well formatted, easy-to-read view. Each column in the log can be enabled or disabled for the test case and provides valuable input for post simulation debug.
PCIe transaction-layer packet logging can also be enabled from a test.

8. Flexibility and reusability

NVMe QVIP can be integrated with PCIe VIP since it uses flexible and configurable generic APIs to interact with PCIe BFM.

These APIs are also protocol independent and allow users to extend default behavior and implement user specific functionalities.

This provides hooks to use for implementing custom behavior or upcoming standards/extensions; such as the NVMe management interface and NVMe over fabrics, etc.

9. Sample functional coverage and compliance test suite

Since coverage driven verification is a natural complement to constrained random verification (CRT), QVIP comes with a sample functional coverage plan and integrated coverage model. It helps in making decisions regarding “are we done yet” with stimuli generation. NVMe coverage matrices are supplied with “must have” and “nice to have” coverpoints and crosses.

Questa VIP is also accompanied with test suite targeting scenarios as specified by industry leading IOL’s NVMe Conformance Test suite v1.2. These sequences are pre-tested and can be used as is within user simulation environments.

CONCLUSION

NVMe is designed to optimize the processor’s driver stack so it can handle the high IOPS associated with flash storage. It also puts the SSD close to the server chipset, reducing latency and further increasing performance while keeping the traditional form factor that IT managers are comfortable servicing. It brings PCIe SSDs into the mainstream with a streamlined protocol that is efficient and scalable and with an ease of deployment through industry standard support.
NVMe QVIP perfectly complements verification engineering efforts for timely design closure by providing easy stimuli generation and debugging interfaces, so they can focus more on protocol-level intricacies.

REFERENCES

1. NVM Express revision 1.2 specification
2. Serial ATA Advanced Host Controller Interface (AHCI) 1.3.1 specification
3. PCI Express® Base Specification Revision 4.0 Version 0.5
4. Register Package guide for UVM, COOKBOOK
8. “The Nonvolatile Memory Transformation of Client Storage”, Amber Huffman and Dale Juenemann, Intel, 2013 IEEE, Published by the IEEE Computer Society
**INTRODUCTION**

The MIPI Alliance signature dishes, C-PHY™ and D-PHY™, are becoming favorite dishes of the imaging industry. These interfaces allow system designers to easily scale up the existing MIPI Alliance Camera Serial Interface (CSI-2™) and Display Serial Interface (DSI™) ecosystems to support higher resolution image sensors and displays while keeping low power consumption at the same time. This gives them an edge to get more into the mobile systems with bigger and better pictures.

“The MIPI C-PHY specification was developed to reduce the interface signaling rate to enable a wide range of high-performance and cost-optimized applications, such as very low-cost, low-resolution image sensors; sensors offering up to 60 megapixels; and even 4K display panels,” said Rick Wietfeldt, chair of the MIPI Alliance Technical Steering Group.

“The MIPI C-PHY, D-PHY and M-PHY®, these three physical layers, combined with MIPI Alliance application protocols, address the evolving interface needs of the entire mobile device. Fundamentally, MIPI Alliance interfaces enable manufacturers to simplify the design process, reduce costs, create economies of scale and shorten time-to-market,” said Ken Drottar, chair of the MIPI Alliance PHY Working Group.

The MIPI Alliance is a non-profit corporation that operates as an open membership organization. All companies in the mobile device industry are encouraged to join, including semiconductor companies, software vendors, IP providers, peripheral manufacturers, test labs and end product OEMs. The MIPI Alliance's charter is to foster standardization by developing a comprehensive set of interface specifications for mobile and mobile-influenced products. MIPI specifications impact both hardware and software in mobile devices. The MIPI Alliance has delivered more than 45 specifications within the mobile ecosystem and is now going beyond mobile devices, finding new applications in vertical segments; such as automotive, IoT (Internet of Things), and wearables.

The MIPI Alliance helps companies interconnect components to create successful mobile designs and provides a selection of physical layer technologies to support a full range of application requirements, yielding benefits across the industry for better alignment on physical interfaces between processors and peripheral devices. The MIPI Alliance intends to reduce fragmentation and improve interoperability among system components, benefiting the entire mobile industry.

**MIPI PHYs**

The MIPI Alliance develops interfaces for mobile device displays and cameras using low power, low noise and high performance physical layer (PHY) serial interfaces. MIPI has defined three physical layer interface devices—D-PHY, M-PHY and C-PHY — and supporting specifications for camera, display, and chip-to-chip protocols. Each physical layer offers unique advantages and features that collectively address every important aspect of today's integrated handheld mobile devices.

These specifications are broadening the variety of interface choices available to manufacturers and opening up new opportunities for companies to differentiate their product designs based on business-specific strategies or technology requirements.

As technology has improved, sensors capture more color information and support more RGB/YCbCr multicolor formats. Extra pixels are required to absorb intense colors. To improve image quality, a system needs improvements in each component and at all the stages of the display ecosystem. MIPI Alliance understands this problem and the associated challenges, offering different
stacks and PHY options for each stage of the imaging ecosystem. This provides faster response time with quick capture and image display. MIPI CSI-2 and DSI uses the first generation MIPI physical layer interface, called D-PHY.

**MIPI D-PHY: GOOD MAN**

MIPI D-PHY had all good qualities, provided simple, scalable solutions and serialized whatever is required. It was a “Good Man.” The D-PHY substantially increases the bandwidth (2.5 Gigabits per lane) for transferring more pixels while consuming very low power. In addition, it maintains a low pin count, which invariably saves power. Until now, everything was running perfectly fine and the industry looked quite satisfied. However, the only thing that is constant is change, and the same applies to the imaging industry as well. Who would have thought that we would move to mega-pixel cameras from VGA cameras? Everyone now foresees more pixel requirements that require much more bandwidth. The immediate solution for this was to use compression techniques. Thus, the MIPI Alliance in collaboration with VESA incorporated DSC compression techniques (MIPI, VESA, DSC). However, bandwidth was still seen as the limiting factor, and the situation arrived where the industry could not go where it needed to go. To resolve this issue, M-PHY was developed— a “Super Man.”

**MIPI M-PHY: SUPER MAN**

MIPI M-PHY has superpowers and fascinating stuff like embedded clocks with clock recoveries, ultra-high bandwidth (5.8 Gigabits per lane), and great performance — all while consuming much less power. This is the next generation PHY that supports a much broader range of applications, including interfaces for advanced displays and cameras, audio, video, memory, power management, and communication between Baseband IC to Radio Frequency IC. The M-PHY has everything that is extraordinary and requires extraordinary stacks, like the MIPI Unified Protocol (UniPro™), to communicate, but the industry is still playing dirty with ordinary stacks, like MIPI CSI-2 and MIPI DSI. The existing architecture of CSI-2 and DSI, including their intermediate revisions, are ordinary and do not allow designers to move to an extraordinary M-PHY for higher speeds from the traditionally used D-PHY. The decision to incorporate MIPI UniPro is not feasible and certainly is not a few months job because it is not an upgrade. Rather, it requires system-level architectural changes, which affects the entire imaging ecosystem and its cost. So, the industry started looking for something new, or at least something different. They started looking towards the ray of light coming through a tiny hole, which looked like a “C”, and then the “Man of the Hour” shone through – MIPI C-PHY.

**MIPI C-PHY: THE MAN OF THE HOUR**

MIPI C-PHY provides the best solution for the OEMs or IP vendors, which are currently using MIPI D-PHY as a PHY layer for their legacy MIPI CSI-2 and MIPI DSI stacks. The C-PHY is giving wings to the imaging ecosystem. The C-PHY satisfies the backward compatibilities with D-PHY and can be implemented on the same parallel interface (PHY Protocol Interface – PPI). The C-PHY uses the 3-Phase symbol encoding technology, which delivers approximately 2.28 bits per symbol over a three-wire group of conductors per lane. This enables higher data rates at a lower toggling frequency, further reducing power. The bandwidth (5.7 Gigabits per lane) is ultra-high in comparison to MIPI M-PHY. The C-PHY can be connected to an ordinary legacy stack (including displays and cameras) of the D-PHY via PPI and will provide the high-speed interface. It holds an embedded clock for HS transfer and clock recoveries. It has been designed specifically for moving pixels from the source to sink without any overhead and with minimal latency, by which it will provide a high-throughput performance. The data width of parallel interface (PPI) that communicates to a camera (MIPI CSI-2) and display (MIPI DSI) stack has been scaled up. Now, more data can be transferred to C-PHY within a single clock. These are the reasons that C-PHY is becoming the favorite and certainly the man of the hour. The MIPI Alliance is also working on upgrading the D-PHY to match the industry requirements.
MENTOR QVIP (QUESTA VERIFICATION IP)

The Mentor Graphics® Questa® Verification IP (QVIP) library provides the MIPI D-PHY, M-PHY and C-PHY VIP for the latest-generation physical layer solutions and protocol specifications, including imaging CSI-2 and DSI for MIPI based ecosystems. Compatible with the Questa simulator and functional verification platform, MIPI QVIPs deliver easy-to-use models and high performance with advanced debug capabilities.

Mentor’s MIPI QVIP integrates seamlessly into users advanced verification environments, including testbenches built using UVM, OVM, Verilog, VHDL, and System-C, and can be very easily applied to a MIPI-based, design-under-test (DUT). The MIPI QVIPs also comes with predefined verification environments, so users can simply plug-and-play their DUT. In this way, the user is up and running in a short period of time even on their first test.

Because the connection between the testbench and the QVIP is at the transaction level, rather than at the signal interfaces, and transaction-based verification gives better control over traffic generation from the lowest to highest abstraction levels covering all possible scenarios, it will be highly effective and could help customers in achieving their verification goals.

All Mentor QVIP delivers the following features:

- Comprehensive test suite and compliance tests
- Complete protocol coverage
- Assertion checking for protocol complexities
- Native SystemVerilog OVM and UVM tests and components
- SystemVerilog, Verilog, VHDL, and System-C testbench support
- Integrated support for verification planning and management
- Transaction-level scoreboarding, analysis, and debug

When beginning a new design, it is common to evaluate how to build a verification infrastructure in the quickest amount of time. Of course, verification has to be complete enough to improve confidence in the design. Rapid bring-up and improving the quality of your design are excellent goals. However, you should not forget that your environment should be efficient to use during the

<table>
<thead>
<tr>
<th>PHY Characteristics</th>
<th>MIPI M-PHY™ v3.1</th>
<th>MIPI D-PHY™ v1.2</th>
<th>MIPI C-PHY™ v1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primary use case</strong></td>
<td>Performance driven, bi-directional packet/network oriented interface</td>
<td>Efficient unidirectional streaming interface, with low speed in-band reverse channel</td>
<td>Efficient unidirectional streaming interface, with low speed in-band reverse channel</td>
</tr>
<tr>
<td><strong>HS clocking method</strong></td>
<td>Embedded Clock</td>
<td>DDR Source-Sync Clock</td>
<td>Embedded Clock</td>
</tr>
<tr>
<td><strong>Channel compensation</strong></td>
<td>Equalization</td>
<td>Data skew control relative to clock</td>
<td>Encoding to reduce data toggle rate</td>
</tr>
<tr>
<td><strong>Minimum configuration and pins</strong></td>
<td>1 lane per direction, dual-simplex, 2 pins each (4 total)</td>
<td>1 lane plus clock, simplex, 4 pins</td>
<td>1 lane (trio), simplex, 3 pins</td>
</tr>
<tr>
<td><strong>Maximum transmitter swing amplitude</strong></td>
<td>SA: 250mV (peak) LA: 500mV (peak)</td>
<td>LP: 1300mV (peak) HS: 360mV (peak)</td>
<td>LP: 1300mV (peak) HS: 425mV (peak)</td>
</tr>
<tr>
<td><strong>Data rate per lane (HS)</strong></td>
<td>HS-G1: 1.25, 1.45 Gb/s HS-G2: 2.5, 2.9 Gb/s HS-G3: 5.0, 5.8 Gb/s (Line rates are 8b10b encoded)</td>
<td>80 Mbps to ~2.5 Gbps (aggregate)</td>
<td>80 Msysm/s to 2.5 Gsysm/s times 2.28 bits/sym, or max 5.7 Gpbs (aggregate)</td>
</tr>
<tr>
<td><strong>Data rate per lane (LS)</strong></td>
<td>10kbps – 600 Mbps</td>
<td>&lt; 10 Mbps</td>
<td>&lt; 10 Mbps</td>
</tr>
<tr>
<td><strong>Bandwidth per Port (3 or 4 lanes)</strong></td>
<td>~4.0 – 18.6 Gb/s (aggregate BW)</td>
<td>Max ~10 Gbps per 4-lane port (aggregate)</td>
<td>Max ~17.1 Gbps per 3-lane port (aggregate)</td>
</tr>
<tr>
<td><strong>Typical pins per Port (3 or 4 lanes)</strong></td>
<td>10 (4 lanes TX, 1 lane RX)</td>
<td>10 (4 lanes, 1 lane clock)</td>
<td>9 (3 lanes)</td>
</tr>
</tbody>
</table>
verification process. This is where you will spend most of your time. Arguably, debugging design bugs is one of the most time consuming tasks of any project. Transaction Level Modeling (TLM) will change the way you think about debug productivity. Transaction loggers, along with the waveform transaction-debug capabilities of the Mentor QVIP reduces the time to debug and helps improve overall verification time and productivity.

CONCLUSION
All the existing imaging designs (MIPI CSI and MIPI DSI) can make use of the new MIPI C-PHY for greater throughput performance over bandwidth-limited channels to maximize the data rate. This will also increase the effective bit rate and does not require a separate clock lane. MIPI C-PHY is responsive to the emerging market requirements and backward compatible to existing designs.
Almost all electronics systems use memory components, either for storing executable software or for storing data. Accurate memory models are fundamental. Making these models available in proven, standards-based libraries is essential to functional verification of these kinds of designs. The models that make up the library should possess specific qualities, and the library itself should deliver a comprehensive solution that supports any type of simulation environment.

High fidelity memory models should include:

- Front and back door memory protocol interfaces
- Assertions
- Functional coverage monitors
- Memory protocol debug support
- Standards compliance
- Compatibility with all major simulators
- Flexible configuration for second-source evaluation

Mentor Graphics® now offers a new, comprehensive memory Verification IP (VIP) library that embodies all of these qualities and addresses the growing need for accurate memory simulation models.

MEMORY MODEL ESSENTIAL

For verification modelling purposes a memory device can be abstracted as a signal-level protocol interface to a storage array. The signal-level interface has to conform to the timing and behavior of the memory protocol, which may be specified in an industry standard, such as the JEDEC JES79-3F standard for DDR3 or, in a specific case, it may be described in a device manufacturer’s datasheet. How the storage array is implemented is not directly visible to the user, but for simulation models it is generally implemented using either a SystemVerilog data structure or an optimized C data structure.

When a memory model is used in a testbench, it is instantiated as a component that is connected to a memory controller, which is either the design under test (DUT) or part of the DUT. Accesses to the memory take place using the signals of the memory model’s protocol front end, and data is transferred in and out of the model’s storage array. The complexity of the front-end protocol varies by memory type, but it can involve concurrent transfers interacting with a memory state space using control and status registers. Getting decent performance from some types of memory, such as DDR, relies on the controller recognizing certain types of data traffic and reorganizing the memory accesses to optimize the access rate in and out of memory. This level of sophistication requires a high fidelity memory model not only to reproduce the complex behavior and timing of a real memory device but also to determine whether the controller optimizations are effective.

The front door access to the memory over the protocol interface is mandatory for functional verification, but it does take time to transfer data in and out of the memory; therefore a backdoor interface is used to directly load or unload the memory storage array. The backdoor interface is typically used to load an executable software image in a memory model, or it may be used to load data content that is going to be manipulated by a hardware accelerator. The backdoor interface can also be used either to check memory content during a test or compare the memory data against a golden reference at the end of a test. A backdoor interface can also be coupled to a memory debugger to allow memory content to be interactively viewed and changed.

As well as providing this useful front and back door functionality, other capabilities are required to enhance the usability of the memory model for verification. The memory model should provide a means of checking that the memory protocol is being followed and flagging errors as they occur. This is usually supported by the use of assertions that are fired when an error occurs, making it easier to get to the root cause of the problem. The memory model should provide a functional coverage monitor that tracks the different ways in which the protocol has been used. This can be used to check that a memory controller has been thoroughly verified, or to understand which modes of protocol operation have not been tested. Support for
debugging the memory protocol is also important in order to efficiently trace the source of a bug.

THE MENTOR MEMORY LIBRARY

The Mentor memory model VIP library contains 25 of the most commonly used memory types. Through configuration, the library supports thousands of models based on memory devices, and users are able to create their own configurations, allowing an almost infinite number of models to be supported.

The timing and behavior of the models are highly configurable allowing them to be tuned to take on the personality of real memory devices. Each memory module has a MANUFACTURER and a PART_NUMBER parameter that allows you to specify which device the model should behave like. These parameters are used at the beginning of a simulation to set up the appropriate configuration options within the model. The value of these part number parameters can be changed as a simulator command line option. This makes it possible to change the part modelled without having to recompile the design and testbench, which is useful for checking that a second source component will work in a system design.

Table 1: Memory model types currently supported by the Mentor memory library.

<table>
<thead>
<tr>
<th>DDR Memory Type Support</th>
<th>Flash Memory Type Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR2, DDR3, DDR4</td>
<td>SDCard</td>
</tr>
<tr>
<td>LPDDR2, LPDDR3, LPDDR4</td>
<td>eMMC</td>
</tr>
<tr>
<td>UDIMM (DDR2, DDR3, DDR4)</td>
<td>ONFI</td>
</tr>
<tr>
<td>RDIMM (DDR2, DDR3, DDR4)</td>
<td>Serial Flash</td>
</tr>
<tr>
<td>LRDIMM (DDR3, DDR4)</td>
<td>NAND Flash</td>
</tr>
<tr>
<td>DFI</td>
<td>NOR Flash</td>
</tr>
<tr>
<td>Wide IO, Wide IO2</td>
<td>UFS</td>
</tr>
<tr>
<td>HMC</td>
<td></td>
</tr>
<tr>
<td>HBM</td>
<td></td>
</tr>
</tbody>
</table>

The memory models are supported by the Mentor VIP configuration GUI, which allows you to either instantiate a specific memory component model in a testbench created by the GUI or create a configuration file that can be loaded by the memory component. The GUI gives you access to all of the configuration options available for a specific memory type, including timing parameters. This allows you to create your own variant of a memory model to explore specific corner cases. The path to the configuration file generated by the GUI is another parameter of the memory module and, if specified, overrides the MANUFACTURER and PART_NUMBER configurations.

All of the models available in the library can be used either as a stand-alone memory model or as a UVM agent, supporting any type of simulation environment. The Mentor VIP models are qualified on all of the three main EDA simulation platforms — Questa® from Mentor Graphics, Incisive® from Cadence®, and VCS® from Synopsys®.

The memory models are packaged as SystemVerilog modules with a pin-out corresponding to the modelled memory type. This allows them to be instantiated as components in a design netlist or in the top level of a test harness. The models respond to the signal-level protocol for front-door accesses and provide a back-door API that is common across the library.

The models provide full functionality and timing accuracy for each type of memory model. This includes optional functional mode settings and support of advanced operations, such as training and levelling, which is used to fine-tune the response of high-speed protocol interfaces, such as DDR4.

The models have an API so you can reconfigure them during a simulation. The API allows either a new device part number to be specified or a different configuration...
file to be used. Almost any change can be made to the model using this approach. However, you must make sure that any change in behavior occurs at a reasonable place in the test, and you must be aware that the model may require re-initialization or training, depending on the extent of the reconfiguration.

Memory models have other module-level parameters that provide optional verification features. A functional coverage monitor for the memory protocol is activated using the ENABLE_FUNC_COV parameter. A memory transaction logger is turned on using the ENABLE_TXN_LOG parameter. The memory transaction logger either writes to the simulator transcript or to a specified log file, with the output being useful for tracing memory-level simulation activity.

Since the memory models are built on top of the Mentor Verification IP architecture, they have a built-in transactional debug facility that allows protocol activity to be viewed in a waveform window along with other design-level signals. This capability abstracts the memory accesses to high-level transactions, making it easy to understand what is going on in the memory protocol at any point in time.

ON THE FLY MODEL RECONFIGURATION

The general use case for the Mentor memory models is as stand-alone models that represent memory devices at the system level. In this case, they model a device’s behavior for the duration of a test case and are not reconfigured. However, there are some memory controller verification scenarios where it is useful to change the timing or behavior of the model during the course of a simulation. This is supported by the Mentor memory models in one of three ways. Major changes to the configuration of the model to change the part number or to reconfigure it to a custom configuration can be made using a run-time API call — either specifying a new part number or the path to a new custom configuration file. Smaller run-time changes can be made by directly setting configuration variables in the model itself.

All of the available configuration variables for each model are documented in the comprehensive on-line documentation available with the library. For instance, making a subtle timing change to check that a memory controller can cope with a late timing response can be done by changing the appropriate timing variable on the fly.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANUFACTURER</td>
<td>Identify part manufacturer</td>
</tr>
<tr>
<td>PART_NUMBER</td>
<td>Identifies device part number</td>
</tr>
<tr>
<td>CONFIG_FILE</td>
<td>The path to a custom configuration file created by the configuration GUI</td>
</tr>
<tr>
<td>ENABLE_FUNC_COV</td>
<td>Enables the functional coverage monitor</td>
</tr>
<tr>
<td>ENABLE_TXN_LOG</td>
<td>Enables the transaction logger for debug</td>
</tr>
<tr>
<td>TXN_LOG_FILE</td>
<td>File path for the transaction logger output</td>
</tr>
<tr>
<td>IF_NAME</td>
<td>Used with uvm_config_db to identify the models virtual interface handle</td>
</tr>
</tbody>
</table>

Table 2: Mentor memory model module parameters.

Changes to the configuration variables can be made using either hierarchical references in Verilog or VHDL testbenches or via a virtual interface handle in a SystemVerilog UVM testbench. The model places a reference to its virtual interface handle into the UVM uvm_config_db data structure, and components in the UVM testbench can reference the model’s configuration parameters this way. Module-level parameters are provided to customize the naming of the path and key strings that are used for storing and retrieving the models virtual interface handle from the uvm_config_db.

CONCLUSION

The Mentor memory model library provides a comprehensive memory modelling solution and comes with an extensive range of configuration options that allow specific device parts or custom parts to be modelled very easily. The models come with built-in advanced verification features, can be used in any form of simulation-based verification environment, and are qualified to run on the Questa, Incisive, and VCS simulators.

Want to learn more about VIP challenges and solutions? Read our whitepapers on various VIP applications, including Verifying Display Standards — A Comprehensive UVM Based Verification IP Solution and DFI with Wide-IO—Delving Into the Unknown Recipe, on Verification Academy.
FPGA PROTOTYPE RUNNING—NOW WHAT?
Well done team; we’ve managed to get 100’s of millions of gates of FPGA-hostile RTL running at 10MHz split across a dozen FPGAs. Now what? The first SoC silicon arrives in a few months so let’s get going with integrating our software with the hardware, and testing the heck out of it. For that, we’ll need to really understand what’s going on inside all those FPGAs.

Ah, there’s the rub.

Our conversations with very many prototypers, confirmed by numerous user surveys, tell us that debug has emerged as just about the biggest challenge for prototypers today. In fact, debug is really a series of challenges.

DEBUGGING A PROTOTYPE IS A MULTI-LAYERED PROBLEM
Assuming you trust your FPGA hardware, the first challenge is to make sure you haven’t introduced new bugs during the process of converting and partitioning the SoC into the FPGA hardware. Any unintended functional inconsistency between the SoC design and its FPGA prototype means that we don’t even get to first base.

The recommended approach is to bring the design up piecemeal, and debug each new piece in its turn. Some prototypes can be driven via a signal-level interface from the RTL simulation testbench to the top-level ports on the design. Then a relevant simulation testbench can be applied as each function is added to the prototype in turn. When the whole design passes this initial check, we can be confident that the FPGA prototype is a valid cycle-accurate representation of our RTL.

We can now get on with using the prototype in earnest, and for this we will need to gain visibility inside the FPGAs using embedded instrumentation.

EMBEDDED INSTRUMENTATION FOR ALL REASONS
We can identify these five tasks during the prototype bring-up and usage for which internal visibility is essential . . .

1) The configuration and integrity of the bare FPGA hardware
2) The piecemeal test of blocks of RTL during bring-up
3) The long-duration testing of the RTL
4) The in-circuit validation of software, including hardware-software co-debug
5) Final system integration

Ideally we want a common debug approach for all these steps. The Visualizer™ Debug Environment from Mentor Graphics®, in combination with Certus™ Silicon debug provides exactly that. Visualizer is a key component of the Mentor Graphics Enterprise Verification Platform™ (EVP), so it may already be familiar to the prototype team, who may have used it in Questa® for simulation and Veloce® for emulation debug earlier in the SoC project. You can read more about Visualizer and EVP in back-issues of Verification Horizons, but for now, let’s focus on how Certus provides the trace data that the Visualizer needs in order to trace bugs to their root cause.

FROM TRIGGER TO ROOT CAUSE CAN BE A LONG JOURNEY
The hardest to find bugs may only become apparent in the prototype many thousands or even millions of clock cycles after their root causes have occurred. These are exactly the kinds of bugs that tend to emerge only at prototype stage. The more trace data our embedded instrumentation can capture, the sooner we can find the root cause.

The root cause may also be in a location far removed from the observed bug, often in a different clock domain, and maybe even a different FPGA on the prototype. A number of different trigger conditions might also be required in order to expose the bug’s erroneous behavior. These challenges give us a shopping list of features we need in our prototype’s debugger . . .

• Wide trace capture across multiple FPGAs
• Extremely long trace depth
• Multiple concurrent and flexible triggers

The latest version of the Certus debugger delivers these features, providing real benefit to prototypers and other FPGA users.
OPEN WIDE

The first benefit of Certus is its ability to choose among 10’s of thousands of internal FPGA signals from a single instrument, called a capture station. This is made possible by the unique observation network shown along with a capture station in Figure 1. An observation network is an optimally efficient non-blocking switching network that funnels the chosen RTL signals to the capture station. The capture station itself contains logic for tracing, compressing and storing activity on the selected RTL signals. The trace width per station can be from 16 to 1024 on binary boundaries, but a typical usage is 256 signals, allowing for multiple medium-sized capture stations in a typically utilized FPGA.

Each station is synchronous with one of the FPGA’s clock domains and has the ability to trace signals from any of those in its observation network inputs, selectable at runtime. This freedom to switch between signals at runtime is a huge advantage for prototypers in the lab.

Typically, prototypers connect the observation network to as many signals as may be even remotely interesting to trace, because all of these will be selectable later in the lab. Traditional debug tools force the user to guess which signals we will need to be traced in advance, way back at the start of the tool flow. Murphy’s Law says that once in the lab, the one critical signal that the user needed to trace has not been selected in advance, so they must return to the beginning and re-instrument. This tiresome loop is much, much less likely to happen with Certus because we can select from 10’s of thousands of available signals at runtime.

As we can see in Figure 2, multiple capture stations can be instantiated in any given FPGA, or any given clock domain in order to extend the capture width even further. Users of traditional embedded debug tools may have been reluctant to add multiple instruments to the same clock domain, since the captured data could not be correlated between the stations.

However, Certus uses its on-FPGA router block to provide inter-station communication, in order to calibrate and synchronize the trace data from each station, and to transfer everything to an external host for assembly into a common, time-aligned database for analysis. We can see in Figure 2 that transfer is usually done via high-speed JTAG, but in the special case of ProFPGA boards, we will achieve even greater speed by using ProDesign’s MMI-64 communications bus.

As FPGA prototypes almost always have to implement multi-clock SoC designs, it is important that this Certus ability overcomes limitations of traditional single-clock, single-FPGA debuggers.

EXTENDING TRACE DEPTH

Typically, embedded FPGA debuggers employ unused internal block RAM to store trace data. However, block RAM is a limited resource in any FPGA, so the maximum possible trace depth may be too short. If so, then users are forced to “walk” successively refined triggers back towards
the root cause event, capturing fresh data at each step. Traditionally, this walking can take several days since each trigger might well require a re-instrumentation back at the RTL. It’s preferable to extend the trace depth either finding more RAM or making better use of the RAM that we have.

Certus makes better use of block RAM employing data compression on the trace data; typically increasing trace depth by up to 1000x over uncompressed approaches. The actual compression achieved is data-dependent but as an example, storing all read and write traffic on an AXI bus to all memory locations in an ARM® A7 design may yield compression of over 1000x. At other times, on the same bus, the compression might be as low as 3x nevertheless that is still hugely significant.

An even better and easier way to extend trace depth is to store the compressed trace data in dedicated external memory, as provided on any good prototyping hardware. Using external memory allows trace data to be captured for thousands of times longer than the upper limit imposed by the available block RAM.

Mentor Graphics has been developing debug technology that uses external memory, enabling the capture of seconds of real time data, at nanosecond resolutions. The capture length depends on the compression, trace width and design clock speed but even a worst case example with no compression, thousands of signals can be captured for 100s of milliseconds. What does that mean in real life? Consider capturing the entire length of an OS boot sequence in order to trace back why a certain driver did not initialize correctly. Alternatively, how about conditionally capturing every occurrence of a particularly rare packet header over many days of test of a network SoC prototype? This is an extraordinary leap forward in FPGA prototype observability.

We call this breakthrough in trace depth streaming, and it will be made available initially to users of DDR3 and Virtex®-7 or DDR4 and UltraScale®, both on proFPGA hardware, with other memory configurations to follow. Figure 3 shows how in streaming mode, the instrumentation logic is extended to include two important Xilinx® IP blocks; a MIG block created from Xilinx’s Memory Interface Generator and an AXI Network-Interconnect (NIC).

Using this configuration, we have already proven that streaming on Virtex-7 and DDR3 hardware will achieve a throughput of over 10 Gbps.

This raw speed allows the streaming technology to keep up with the design running in the FPGAs by extracting trace data at a rate which is typically 15x faster than the prototype’s highest system clock rate. Each capture station’s internal memory acts as a FIFO buffer to provide further data elasticity, and data compression brings further benefit.

It is even possible that streaming data can be sent directly to the host via a fast connection, such as ProDesign’s MMI-64 bus. In a recent application, streaming was able to capture all reads and write activity to system RAM (a trace width of approximately 500 signals) on an ARM®-A7 prototype during its entire three minute boot-up of Linux.

As another boon to prototypers, each capture station can be configured independently to store its trace data in local block RAM or in external RAM, or stream directly to host. Then at runtime, each streaming-capable capture station can be configured to either stream data or to store data in its local block RAM.

**CALLING ALL STATIONS; THIS IS AN ALL FPGA ALERT!**

The last of our three identified major requirements for prototype debug is trigger flexibility and concurrency across all FPGAs. We have noted that trace data in the various capture stations are kept in sync by inter-
station communication and the router, but let’s see how communications are extended across multiple FPGAs to enable complex whole-system triggering.

The latest release of Certus uses enhanced communications links between the routers, called event channels. These event channels can be seen operating between FPGAs in Figure 4.

First, let’s recap on Certus triggering. A capture station can be set up at run-time in order to continuously capture data until a certain condition on its inputs is met. This is the trigger condition, which can be any logical combination of the observed signals. Think about that; any input to the observation network can be used as either trigger or trace (or both) so we avoid the long iterations which plague other debug tools that force the user to guess the inputs to the cone of trigger logic too early in the flow.

We can set up different trigger conditions in each capture station so that if one of them occurs a trigger event is propagated over the event channels so that the ongoing trace capture is stopped effectively simultaneously in all stations, and the data in the Visualizer display will be correctly aligned in time.

The event channel can be used not only for triggers but also to alert all other capture stations that an event has occurred. This allows more complex triggers and conditional tracing to be employed. For example, one capture station might look for an initial condition, which becomes an event, then other stations can be set up to look for that event before enabling their own trigger conditions. Don’t forget, these stations might be in different FPGAs, running on different clock domains, observing different signals. This is useful for FPGA prototyping because contiguous blocks in the SoC design are often partitioned across multiple FPGAs, but still need to be debugged as one function.

**REAL DEBUG FOR FPGA PROTOTYPES**

Traditional FPGA debuggers might be acceptable for single FPGA designs, or where long instrumentation iterations might somehow be tolerable, but they run out of steam when faced with a full-scale, FPGA prototype of today’s complex SoCs. Certus Silicon Debug is ready to take up the baton and provide prototypers with the visibility and productivity they demand.

To take a closer look at Certus, please visit www.mentor.com/certus, contact your local Mentor representative or if you want to discuss your FPGA debug challenges, feel free to contact me at doug.amos@mentor.com.

![Figure 4: Inter-FPGA communication using Certus event channels](image-url)
INTRODUCTION
DO-254 and other safety critical applications require meticulous initial requirements capture followed by accurate functional verification. “Elemental Analysis” in DO-254 refers to the verification completeness to ensure that all ‘elements’ of a design are actually exercised in the pre-planned testing. Code Coverage is good for checking if implementation code has been tested, but cannot guarantee functional accuracy. Currently, functional accuracy is guaranteed using pre-planned directed tests, auditing the test code and auditing the log files. This is not scalable as designs get complex. In this article we will look at using SystemVerilog syntax to concisely describe the functional coverage in the context of accurate “elemental analysis”.

“Safety Specific Verification Analysis” as called out in DO-254 Appendix B addresses the need to check not only intended-function requirements, but also anomalous behaviors. It is left up to the applicant to propose a method to sufficiently stimulate the “element” to expose any anomalous behavior. In this article we shall focus on this “Safety Specific Verification Analysis” or “Sufficient Elemental Analysis”. We will be using UVM to describe the stimulus space to look for anomalous behavior.

The article will also highlight items to use for auditing a flow from DER’s point of view.

VERIFICATION COMPLETION CRITERIA
The key stake holders for a FPGA (or ASIC) Verification include DER, Managers and Engineers. The DER is responsible to audit the flow with the focus on design safety. The managers are trying to meet the customer requirements with shorter design schedules and optimal resources. The designers want to accurately capture the design intent using the tools provided.

The purpose of verification is considered to be a “supporting process” in a DO-254 program. It is not a specific phase of development, but rather occurs throughout the design flow from the earliest models to the final testing of the component in the system. The primary objective is to ensure that a design performs the function specified by its requirements and that it satisfies agreed-upon completion criteria. Safety critical DAL A and B devices require that the verification be carried out independent of the design.

“Elemental Analysis” in DO-254 refers to the verification completeness to ensure that all ‘elements’ of a design are actually exercised in the pre-planned testing. An ‘element’ is the smallest design item that an engineer uses to create the design. In the context of VHDL/Verilog language based design of FPGA/ASIC devices this can vary from a statement to a conditional block, or some other larger structure such as a reusable internal/external IP, or a reusable bus interface protocol.

Code Coverage is a good metric for determining if the design implementation code statements, conditional blocks or FSM coding structures are exercised by simulation. But Code Coverage will not assure the functional accuracy, integrity of a reusable IP block or a bus protocol. This leads to writing too many redundant tests and wasted simulation cycles. See Figure 1. Did the higher Code Coverage mean the design is functionally accurate? Did the manual audit miss functionality? Did the larger design state space get exercised to look for any anomalous behavior? Is there unnecessary testing wasting simulation cycles?

When planning for sufficient elemental analysis at every stage of the design flow one needs to confirm things identified in Table 1.

Figure 1: Types of Tests
DERs would like a tool that showed the spec in an executable format for stimulus and response. This can reduce the time currently spent on manual audit. Managers would like to eliminate unnecessary redundant waste of simulation cycles and designer's time. And designers would like to use the latest methodology to make their delivery more reliable and robust.

The testing needs to be pre-planned to cover the required modes and configurations, but it also needs to make sure other unused configurations and modes have pre-determined safe behavior. SystemVerilog can be used to capture the external stimulus/response spec in a readable and executable format. To guide the stimulus we can use Functional Coverage. It is also important to have a means of sufficiently testing the larger design state using a vast range of scenarios beyond the pre-planned “directed” necessary tests. The SystemVerilog constructs are made much more consistent to use via UVM techniques.

**UVM (UNIVERSAL VERIFICATION METHODOLOGY)**

UVM is an open-source library of SystemVerilog language-based source code maintained by Accellera.org. It is developed by a conglomeration of companies and independent developers. It is based upon various precursors of the technologies including proprietary technologies developed and deployed for complex hardware verification over the decades. It is supported by all the major simulation tool vendors and is widely exercised by thousands of projects across the industry. There is a large ecosystem building around UVM to provide code development editors, verification IP, debuggers, and UVM-conversant college graduates.

UVM testbench has the similar goals of applying test vectors and measuring the response as shown in Figure 1.1.

**Table 1: Verification Completion Criteria**

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Stimulus</th>
<th>Functional Metrics</th>
<th>Questions to answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-planned tests</td>
<td>Apply specific stimulus</td>
<td>Spec Coverage</td>
<td>Test delivered Functionality</td>
</tr>
<tr>
<td>Robustness within a given range</td>
<td>Apply wide range of stimulus</td>
<td>Range Coverage</td>
<td>Test accuracy in a range of test cases</td>
</tr>
<tr>
<td>Robustness outside the expected range</td>
<td>Apply range of error-stimulus</td>
<td>Error-case Coverage</td>
<td>Test design tolerance to errors</td>
</tr>
<tr>
<td>Corner cases</td>
<td>Apply corner–tests beyond</td>
<td>Corners Coverage</td>
<td>Exercise corner behavior</td>
</tr>
</tbody>
</table>

DERs would like a tool that showed the spec in an executable format for stimulus and response. This can reduce the time currently spent on manual audit. Managers would like to eliminate unnecessary redundant waste of simulation cycles and designer’s time. And designers would like to use the latest methodology to make their delivery more reliable and robust.

The testing needs to be pre-planned to cover the required modes and configurations, but it also needs to make sure other unused configurations and modes have pre-determined safe behavior. SystemVerilog can be used to capture the external stimulus/response spec in a readable and executable format. To guide the stimulus we can use Functional Coverage. It is also important to have a means of sufficiently testing the larger design state using a vast range of scenarios beyond the pre-planned “directed” necessary tests. The SystemVerilog constructs are made much more consistent to use via UVM techniques.

**UVM (UNIVERSAL VERIFICATION METHODOLOGY)**

UVM is an open-source library of SystemVerilog language-based source code maintained by Accellera.org. It is developed by a conglomeration of companies and independent developers. It is based upon various precursors of the technologies including proprietary technologies developed and deployed for complex hardware verification over the decades. It is supported by all the major simulation tool vendors and is widely exercised by thousands of projects across the industry. There is a large ecosystem building around UVM to provide code development editors, verification IP, debuggers, and UVM-conversant college graduates.

UVM testbench has the similar goals of applying test vectors and measuring the response as shown in Figure 1.1.

**Table 1: Verification Completion Criteria**

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Stimulus</th>
<th>Functional Metrics</th>
<th>Questions to answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-planned tests</td>
<td>Apply specific stimulus</td>
<td>Spec Coverage</td>
<td>Test delivered Functionality</td>
</tr>
<tr>
<td>Robustness within a given range</td>
<td>Apply wide range of stimulus</td>
<td>Range Coverage</td>
<td>Test accuracy in a range of test cases</td>
</tr>
<tr>
<td>Robustness outside the expected range</td>
<td>Apply range of error-stimulus</td>
<td>Error-case Coverage</td>
<td>Test design tolerance to errors</td>
</tr>
<tr>
<td>Corner cases</td>
<td>Apply corner–tests beyond</td>
<td>Corners Coverage</td>
<td>Exercise corner behavior</td>
</tr>
</tbody>
</table>

DERs would like a tool that showed the spec in an executable format for stimulus and response. This can reduce the time currently spent on manual audit. Managers would like to eliminate unnecessary redundant waste of simulation cycles and designer’s time. And designers would like to use the latest methodology to make their delivery more reliable and robust.

The testing needs to be pre-planned to cover the required modes and configurations, but it also needs to make sure other unused configurations and modes have pre-determined safe behavior. SystemVerilog can be used to capture the external stimulus/response spec in a readable and executable format. To guide the stimulus we can use Functional Coverage. It is also important to have a means of sufficiently testing the larger design state using a vast range of scenarios beyond the pre-planned “directed” necessary tests. The SystemVerilog constructs are made much more consistent to use via UVM techniques.

**UVM (UNIVERSAL VERIFICATION METHODOLOGY)**

UVM is an open-source library of SystemVerilog language-based source code maintained by Accellera.org. It is developed by a conglomeration of companies and independent developers. It is based upon various precursors of the technologies including proprietary technologies developed and deployed for complex hardware verification over the decades. It is supported by all the major simulation tool vendors and is widely exercised by thousands of projects across the industry. There is a large ecosystem building around UVM to provide code development editors, verification IP, debuggers, and UVM-conversant college graduates.

UVM testbench has the similar goals of applying test vectors and measuring the response as shown in Figure 1.1.

**Table 1: Verification Completion Criteria**

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Stimulus</th>
<th>Functional Metrics</th>
<th>Questions to answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-planned tests</td>
<td>Apply specific stimulus</td>
<td>Spec Coverage</td>
<td>Test delivered Functionality</td>
</tr>
<tr>
<td>Robustness within a given range</td>
<td>Apply wide range of stimulus</td>
<td>Range Coverage</td>
<td>Test accuracy in a range of test cases</td>
</tr>
<tr>
<td>Robustness outside the expected range</td>
<td>Apply range of error-stimulus</td>
<td>Error-case Coverage</td>
<td>Test design tolerance to errors</td>
</tr>
<tr>
<td>Corner cases</td>
<td>Apply corner–tests beyond</td>
<td>Corners Coverage</td>
<td>Exercise corner behavior</td>
</tr>
</tbody>
</table>

DERs would like a tool that showed the spec in an executable format for stimulus and response. This can reduce the time currently spent on manual audit. Managers would like to eliminate unnecessary redundant waste of simulation cycles and designer’s time. And designers would like to use the latest methodology to make their delivery more reliable and robust.

The testing needs to be pre-planned to cover the required modes and configurations, but it also needs to make sure other unused configurations and modes have pre-determined safe behavior. SystemVerilog can be used to capture the external stimulus/response spec in a readable and executable format. To guide the stimulus we can use Functional Coverage. It is also important to have a means of sufficiently testing the larger design state using a vast range of scenarios beyond the pre-planned “directed” necessary tests. The SystemVerilog constructs are made much more consistent to use via UVM techniques.

**UVM (UNIVERSAL VERIFICATION METHODOLOGY)**

UVM is an open-source library of SystemVerilog language-based source code maintained by Accellera.org. It is developed by a conglomeration of companies and independent developers. It is based upon various precursors of the technologies including proprietary technologies developed and deployed for complex hardware verification over the decades. It is supported by all the major simulation tool vendors and is widely exercised by thousands of projects across the industry. There is a large ecosystem building around UVM to provide code development editors, verification IP, debuggers, and UVM-conversant college graduates.

UVM testbench has the similar goals of applying test vectors and measuring the response as shown in Figure 1.1.
The major difference with UVM is that the structure of UVM testbench is component-based and uses improved object-oriented syntaxes added in the SystemVerilog language. This allows plug-n-play and reuse as shown in Figure 1.2.

Let’s look at some of the key aspects of the UVM testbench using a simple example. We will also highlight how these aspects help a DO-254 type of project.

The design we chose as an example is a Floating-point Unit (FPU) as shown in Figure 2, available to download from OpenCores.org. As seen in the figure, the total number of input bits including the operands and operations is 69-bits. From a testing perspective we cannot possibly test all the 69-bit input data bit combinations ($2^{69}$). The IP comes with a VHDL testbench. It uses 100,000 test cases for each operation and rounding mode amounting to about two million stimulus vectors. The input and expected vectors are provided via a text file. It was followed by hardware testing. But is this IP fully verified?

We will approach the verification using UVM. The primary verification requirements for the FPU are as follows:

- Verify all the five operations and four rounding modes
  - 20 combinations in all
- Verify 32-bit IEEE-754 compliant floating point data inputs
- Verify that exceptions get flagged correctly

UVM testbench development starts with the interface declaration, which is just a collection of DUT pins as shown in Figure 4. This collection of pins is very similar to a module declaration in Verilog. It can also include the assertions to verify timing activity around the interface, such as the number of pipeline delays needed to complete each operation. In this case we want to make sure of the following items:

- Add/Sub operations takes seven cycles
- Multiply operation takes 12 cycles
- Division/square-root operation takes 35 cycles

SystemVerilog assertion syntax allows clock-based timing checks plus data capture at various clock points.

This interface can be bound to a VHDL DUT and VHDL testbench to add the notion of functional coverage without modifying the code. But our focus here is to explore UVM.
Figure 4: Interface with DUT pins and Assertions to catch good and bad timing behavior

The next step is to build capture the stimulus description. This is done by defining transactions. The transaction contains two operands, operation and rounding mode. Transactions are defined at higher level and never talk about control signals or the timing of the bus interaction. The operands are 32-bit floating point numbers. To limit the testcases plus perform exhaustive verification we break down the 32-bit space into 12 ranges as shown below.

1. Positive and Negative zero
2. Positive and Negative Denormalized Real Number with exponent=0
3. Positive and Negative Normalized Real Number with exponent > 0
4. Positive and Negative Infinity
5. Positive and Negative Quiet NaN (Not a Number)
6. Positive and Negative Signaling NaN

This reduces the total stimulus combination to be 12 types of operand A, 12 types of operand B, four operations and five rounding modes. This amounts to a total of 2880 stimulus combinations. We are in essence building stimulus knobs and also response meters as shown in Figure 5.1.

Transactions are defined in UVM using SystemVerilog “class” syntax. This is very similar to VHDL record to encapsulate data but SystemVerilog “class” further allows extension of a type. Each class can have attributes. SystemVerilog “constraints” allow defining the valid range of values for those attributes. See Figure 5.2 on the following page for details on the operand “class” declaration. The attributes have a range of suitable values defined inside “constraints”. It currently only shows Normalized and Denormalized type of operands. This file can be made to exactly align with the stimulus specs and can be used for auditing the ranges defined. Other ranges can be declared using similar syntax.

The UVM Sequence Item or the transaction that will stimulate the DUT is a collection of the operands and operations. It will be used to build the sequences or the testcases as shown in Figure 5.3 on the following page. Notice how the fpu_request declares the class as an extension of uvm_sequence_item, which is part of the UVM library and comes with various utilities and extensible functionality. Also, notice how the fpu_response extends the request transaction and inherits the fpu_request attributes plus adds its own attributes. Inheritance is one of the key features that helps reuse.

The UVM Sequences are a string of transactions. IP-provided VHDL testbench used a C-program generated equivalent Knobs and Meters.
Inheritance allows a parent to share a functionality or a method to its child. The "body()" function is the main built-in method that generates the transactions. This is overridden by the user code which defines the order of generation of sequence items. This can prove as a better way to document the testcases for DO-254 and can prove as a better artifact than two million VHDL test vectors.

Sequences use the knobs we have built earlier. The randomize() function invokes the simulator-built-in constraint solver that solves all the constraints to pick a solution. The "randomize() with {...}" allows further tightening the constraints. If there are any constraint conflicts the function will return "0" and the "assert" around the randomize() call will catch the failure. At the end of the randomize() call the "rand" attributes within the class will have a value picked from the constraints-solved solution space. Each new seed specified picks a new testcase from the space.

The UVM structural blocks that drive the transactions onto the interface (onto the DUT) and independently monitor the bus activity are shown in Figure 7.

The UVM Sequencer reads the sequences we have created. It is the simplest component created as shown in Figure 7.
The **UVM Driver** reads the sequence items generated by the Sequencer (via the chosen sequence). The driver accesses the FPU pins via “virtual interface” and provides the cycle-accurate bus pin wiggling on the DUT. The “run_phase” is the built-in uvm_component “virtual method” that gets invoked or orchestrated by the UVM base code and is customized by the user as shown here. Notice also the use of “forever” loop inside “run_phase” task that extracts the data generated by the sequence via the built-in “seq_item_port.get(m_request)” call. The component’s run_phase() tasks are called as the threads that get launched simultaneously and typically coded to run forever. Sequences typically control the simulation runtime until no items are generated and all threads are killed by UVM base code.

The **UVM Monitor** independently monitors for requests and responses for broadcasting the data to the rest of the testbench. Notice the analysis_port.write() function invocation which is the way to broadcast the observed transaction. Here is where one could also integrate a way to print out stimulus vectors applied and response seen in a file for DER Audit purposes. We will see next how the listeners or subscribers respond to this call.

We are skipping the details on how the UVM components are hooked up and or the details on UVM Agents, UVM Environment and UVM Test. The goal here is to architect the transaction to drive proper stimulus, see it as a documentation artifact. One can also review the bus interface, the corresponding driving/monitoring functionality, if needed.

The **UVM Subscriber component**, as shown in Figure 8 and Figure 9 is a listener to the monitor’s analysis_port.write() function call that broadcasts the observed response transaction. Subscribers receive
the transaction by implementing "write()" callback function.
More than one subscriber can be added to listen to a single monitor analysis port.

The UVM Scoreboard listens to the stimulus and responses. It takes the applied stimulus and predicts the expected output. The prediction algorithm can be C. Scoreboard also compares the expected response with the actual response. See Figure 8. Also, notice the consistent way of message logging.

We can integrate coverage collectors as listeners or subscribers as well. For example, one of the checks required is to check sequential FPU operations – such as ADD followed by SUB, ADD followed by MUL, etc. We have five total operations. This makes it 25 possible sequential operation combinations. This is done via SystemVerilog covergroup syntax that allows transition coverage syntax.

This covers an introduction to some of the essential aspects of UVM. There are other structural aspects of the UVM testbench such as an agent, an environment, and a test. For further details please look at the UVM Cookbook or VerificationAcademy.com.

See Table 2 below for a recommendation on the specific artifacts to be used by DER for a review of a UVM testbench.

### SUMMARY OBSERVATIONS

Code Coverage is one of the recommended and required tools in DO-254 flows. The verification items identified during the course of the example description earlier cannot be covered via Code Coverage, the metric that helps measure these types of verification items is called Functional Coverage. The traditional way to test this would be to write specific testcases for each of the functional scenarios. The result is printed out in a log file which

<table>
<thead>
<tr>
<th>Tests to Audit</th>
<th>Review Source Code</th>
<th>Review Results</th>
<th>Review Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-planned tests</td>
<td>Constraints</td>
<td>Stimulus Vectors</td>
<td>Code Coverage</td>
</tr>
<tr>
<td>Robustness test</td>
<td>Sequences</td>
<td>Response Vectors</td>
<td>Functional Coverage</td>
</tr>
<tr>
<td>(within range)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Robustness test</td>
<td>Functional Accuracy Coverage</td>
<td>Run log files</td>
<td>Robustness Coverage</td>
</tr>
<tr>
<td>(outside the range)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corner tests</td>
<td>Protocols in Driver/Monitor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Important files from DER Audit point of view
is meticulously reviewed. This does involve extra code and extra time to review. Functional Coverage is typically measured via assertions and covergroup syntax in SystemVerilog. Assertions keep track of temporal timing diagram based activity. Covergroups can keep track of data items observed such as types of operations or which of the 12 buckets of the data the operand has hit. We can use CoverGroup and Assertions directly along with VHDL testbenches.

When the assertions and covergroups were applied to VHDL testbenches it was noticed that even 2000 random vectors gave the same coverage as two million vectors. This highlights the fact that more tests doesn’t mean new items are being tested – it merely is repeating redundant tests, which in turn means wasted simulation cycles and wasted productivity. In our measurement we also saw that only a small set of the 2880 combinations was hit by the stimulus which means the stimulus space was not explored intelligently. Adding Functional Coverage provides the visibility on verification efficiency.

When we applied Functional Coverage using assertions and covergroups to the existing VHDL testbench (without modifications or adding UVM) we noticed the following:

• 2000 and up to two million test vectors only achieved about 75% coverage
• Out of the optimal 2880 input vector combinations only about 5% were tested
• Out of 25 sequential operations only 40% were tested
• We noticed three out of five pipeline delay properties fired highlighting bugs in the design or the spec

UVM provides a better structure to deploy SystemVerilog Assertions and CoverGroups. We noticed that with UVM testbenches the verification coverage was achieved faster. This allowed us to do further exploration leading to finding a bug in the design. Here are the observations with UVM:

• We wrote five sequences ran them with two different seeds amounting to 10 tests in total
• We got about 97% functional coverage with 10 tests and 1/10th amount of time
• We covered 100% of the sequential combinations of the operations
• We found a bug with square root operation

In conclusion, two million testcases in VHDL testbench sounded good but SystemVerilog Functional Coverage provided quantification. This article was intended to cover an introduction to some of the essential aspects of UVM, cover how it can help DER auditing process plus how the design/management team can benefit. The article should also provide some hint on the mindset needed for deploying UVM.

REFERENCES:
1. UVM: http://accellera.org/downloads/standards/uvm
2. Floating-point Unit VHDL Design: http://opencores.org/project,fpu100
3. Floating-point Standard: IEEE 754
4. Effective Verification for DO-254, David Landoll, MAPLD 2008
Building a complex signal processing function requires a deep understanding of the signal characteristics and of the different algorithms and their performances.

When it comes to the verification of such designs, a quite generic approach consists of injecting more or less realistic stimulus and using reference models (most often C or Matlab®), to compare the expected results.

Requirement based designs following a DO-254 process add the constraints that each function of the design should be specified as a traceable requirement. The proof of the complete verification of each requirement should also be provided with additional emphasis on physical verification, therefore running the tests on the physical device.

This article describes a combined requirement and metric driven methodology developed at a customer site for the verification of a complex signal processing SoC block under DO-254 constraints. This methodology also enables both horizontal and vertical reuse of the tests, allowing tests to run both in IP simulation and on FPGA boards at SoC level. This approach is described in a generic way and can be applied to different signal or data processing designs.

INTRODUCTION

In the process of building a complex signal processing block, a mathematical model of the function is usually developed to validate the core functionality. In our case, the model was first developed with Matlab and Simulink® and was validated as being a data sample accurate reference model.

As this algorithm completely defines our signal processing function, it would be ideal to describe this algorithm as one big requirement. However, given the complexity of such function and the billions (if not infinite) possible design states and combinations, the high level of abstraction and the algorithm complexity, this high level requirement does not satisfy the unitary-complete-consistent-traceable-unambiguous-verifiable-atomic requirement definition of a “requirement based” approach. We therefore need to split this main function into smaller derived requirements, each of them specifying a specific part of the algorithm (e.g., FIR filter, Correlator, etc…)

The design we had to verify (Figure 1) contains over 50 of such atomic derived requirements, which together specify completely the processing algorithm, from sampling to results in RAM.

Additional to these signal processing functions, dedicated blocks and control logics ensure the communication with the CPU and the systems via interrupts, control and status registers and two AXI4 interfaces. At SoC level, the core CPU can then read and interpret the results and take appropriate actions.

The following sections describe a metric driven and requirement based verification strategy allowing reusable
tests from IP simulations to the implemented SoC on a FPGA board.

**METRIC DRIVEN REQUIREMENT BASED VERIFICATION**

As said above, each atomic computation part of the signal processing algorithm is described as an atomic requirement that the final design shall obey and that the verification shall ensure.

The following example shows a requirement of a FIR filter:

In the above requirement, the use of defined terms have simplified the specification, but this does not make the requirement simpler. The complexity described in the used glossary of terms is actually the key points of the verification analysis.

In this example, the FIR filter is not just a sum, it’s a sum of complex products between complex values in the form of $A = (a + j \cdot b)$ where $j$ is the imaginary number for which $j^2 = -1$.

The verification analysis therefore requires covering additional points:

- Has each complex product been used with minimum, null, maximum values of imaginary and real parts of the sample $X_i$ crossed with minimum, null and maximum values of imaginary and real parts of the coefficient $C_i$?
- Have we reached the worst case where the carry of each single sum has propagated to the final result? On each imaginary and real parts?
- Etc...

This analysis leads to what needs to be covered to actually verify the requirement and can easily be mapped to functional coverage points and coverage groups in SystemVerilog as explained in [3].

In our case, the testbench is architected around a SystemVerilog wrapper, directly monitoring the internal signals of the design described in the requirements (and only these signals) and build covergroups based on the monitored signals (Figure 3 on the following page).
REQUIREMENT BASED ASSERTIONS AND CHECKERS

Although the final test results are compared against the reference model, DO-254 certification will need a clear traceability about the requirements and their verification. If the coverage above ensures that the different conditions of requirement have been exercised, there is no direct proof that the design behaves as specified by each requirement.

Each requirement therefore needs either to be modeled or to rely on assertions to check the required values against the specified expectation.

Adding SVA assertions and in some case small functional SystemVerilog models of the different filters has fulfilled this need. Then using reporting techniques similar to the one described in \(^{(1)}\) allow us to capture and map the assertion results to the requirements.

Note though that since these assertions are actually verifying the requirements, they need additional qualification\(^{(c)}\) effort.

COVERAGE, ASSERTIONS AND TESTS

While the coverage is recording if a requirement has been exercised and the assertions are actually verifying the requirement (providing the fact the coverage is reached), the way to hit our coverage goals requires some more complex testing. Different papers, including\(^{(1)}\) suggest that coverage random methodologies may be applied to DO-254 projects, but providing interesting use cases using such an approach is hard to develop and time consuming for these kinds of complex signal processing blocks. Since we had a Matlab model available, it is much more convenient to use this model to generate our test scenarios.

REFERENCE MODEL AND REQUIREMENTS

The reference model which has served to validate the high level function is architected between a generic core model, and chosen dedicated hardware parameters (size of vectors, filters, width of signals, …).

Adding use case parameters on top of this, the model then holds all the necessary data to generate our test inputs and the expected result of the complete processing. We can therefore compare the result of the RTL with the result of this model to verify the overall functionality of the design, from the input samples to the final result.

As the model has been validated using the same parameters and samples we can therefore ensure that the design follows the same algorithm. Additional to the previous requirement based coverage and assertions, this additional check validates the complete algorithm as a whole and is an additional assurance that the set of requirements is describing the complete algorithm.

In our case, we made the choice to generate the test and its expected results directly from this Matlab model. Generating the test directly from Matlab has the advantage to reduce the number of file processing such as intermediate parsing, but requires some extra Matlab scripting effort.

Other possible approaches could be:

- Link Matlab to the simulator, but this can only work in simulation and not on the physical device
- Use different scripting languages to interpret Matlab results. This would require extra qualification\(^{(c)}\) of the environment
With these, in the end, the generated tests embed:

- The test configuration sequence
- The self-checking function responsible for the result analysis

Several types of implementation are possible for the tests:

- Stimulus command file, but this lacks flexibility, reusability and maintainability in the long run
- UVM sequences
- SystemVerilog tasks or VHDL procedures

SEAMLESS MODULE VERIFICATION

Thinking of how to replay the RTL tests on the FPGA platform, generating C is actually a more reasonable choice:

- In simulation, the C test running on the host computer is using DPI interface to control the AXI4 agents (monitor driver), connected to the design.
- On board level, the C test running on the platform CPU is accessing the hardware directly, from the same software drivers.

The diagram below shows the overall simulation verification environment.

SOC LEVEL PHYSICAL REUSE

Providing some precautions in the test software architecture, we can then remap the write/read register functions and memory DPI calls to direct pointer accesses aligned on the SoC address map, the same C can therefore be reused at SoC level and run bare metal on the physical hardware (see Figure 5 on the following page).

In order to actually be able to reuse the same expected data, the same set of samples needs to be driven to the design. On the physical platform, this could be an issue and may require dedicated hardware prototype, signal generator or a synthesized Bus Functional Model (BFM) to achieve this. Our case was simplified with the use of an embedded signal generator for which the Matlab model was also available in this flow.

Therefore, the main distinctions between the simulation at IP level and the physical device execution is the latency introduced by the software execution from the core CPU on the device (while in simulation, C executes virtually in 0 cycle in regards to the RTL). Another difference is that the physical device testing lacks observability points, assertions, and functional coverage and are therefore missing from the physical testing. But the tests being self-checking, we can consider this self-check as a signature which ensures that the physical device has actually hit the same functions in the same way. Functional coverage of the simulation is therefore hit by the physical tests in the same way.

REQUIREMENT VALIDATION

While the top level algorithm is validated by independent implementations and comparisons of the final expected results, the set of the obtained derived requirements should also be validated.
In particular, the completeness of the derived set of requirements impose that:

- Requirements define outputs in regards to inputs and history
- Each input is either a primary input of the design, or the output of another requirement
- Each output value or sequence is either fully defined, or defined at a specific point in time or on specific conditions
- Special care is taken in regards to outputs/inputs that are valid at specific points in time to ensure they are sampled at the right time by other requirements

A traceability matrix with inputs, outputs and links to the source requirement will validate this completeness.

**CONCLUSION**

The approach described here is combining different verification methodologies. It brings metric driven verification to the requirement based verification. It brings both vertical and horizontal verification reuse from IP level simulation to physical SoC device using a C generation of self-checking tests. Providing functional cover group and assertion traceability up to the requirements, this approach
can be applied to safety critical processes such as DO-254 and ISO 26262.

**TERMS**

The terms "validation", "verification" and "qualification" used in this article are aligned on the DO-254 / ED-80 terminology.

a. Verification ensures that the hardware is what has been specified  
b. Validation ensures the specification is what we want  
c. Qualification ensures that the verification environment and tools are capable of doing verification  

ED-80, Chapter 6, Validation and Verification Process  
ED-80, Chapter 11, Tool Assessment and Qualification

**REFERENCES:**

1. RTCA/DO-254 (EUROCAE/ED-80) Design Assurance Guidance for Airborne Electronic Hardware”  
As per the DO-254 standard, the Airborne Electronic Hardware (AEH) needs accurate assurance of device behavior as intended within optimal operating conditions. For DAL A (Design Assurance Level A) Devices, you need to verify 100% functionality of the device and achieve 100% code coverage, including FEC. Code coverage can be managed using a simulation tool such as Questa®, while functional coverage would require a comprehensive Verification Case Document (VCD) that has cases traced to each requirement of the AEH device.

Once the VCD is defined, it is necessary to ensure accuracy of test bench codes and test cases written to achieve intended operation. To ensure the same, a Verification Procedure Document (VPD) is generated, which consists of test procedures and coverage information for each of the test cases.

Once VCD and VPD are defined, a test plan is generated through linking of both documents, which would prove that all test scenarios are covered with 100% coverage of cover groups, checkers, assertions and test cases. Simulation tools like Questa provide support for generating Test Plan documents. By input of UCDB file using vcover command to Questa, the tool provides the information about the overall coverage and what is not covered. A sample of the Test Plan is shown in Figure 1 below:

Depth of information covered under Test Plan depends on details received from VCD and VPD. Creating a thorough VPD document is a time-consuming and tedious job. In addition to that, it needs to be maintained, with consistent updates into the VPD document so that both sources remain in sync. For example, if any cover groups/assertions are renamed or added, the same must be updated in VPD.

With rising complexities in the AEH designs, such manual processes won’t prove productive and hence risk losing time-to-market. This article gives a simple and fast way of creating such time-consuming and tedious documents using automation.

**AUTOMATED VPD AND TEST PLAN GENERATION**

Different tools accept testplan XML as an input argument, and this XML can be created by simply saving XLS file as a XML, provided we save it in the correct format expected by the tools. Questa from Mentor has an XML2UCDB command for generating detailed testplan coverage reports sourced from the XML files. Following is the example:

```shell
xml2ucdb -debug -verbose -format Excel
-dofilename path.do -datafields "Section,
Title,Description,Path,Link,Type,Weight,Goal,ReqtVerified" testplan.xml UCDB_file
```

In the example above, testplan.xml can be generated by saving an XLS file having the format as shown in Figure 1.

An automated Testplan generation requires the following three steps:

1. Creating Verification Case Document (VCD)
2. Script for extracting testbench and coverage information and to create VPD document
3. Script to create testplan XLS

Step 1 is a manual process and one needs to create VCD based on the requirements with the list of all the test scenarios. In steps 2 and 3, the scripts are required to be generated and users can maintain a certain style of the code while coding test bench and tests, so that the required information can be easily extracted through the script. Nowadays, mostly UVM/OVM methodologies are used which makes steps 2 and 3 easier and faster.

---

**Figure 1: Test plan XLS sheet example**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Section No</td>
<td>Section</td>
<td>Description</td>
<td>Instance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4162</td>
<td>ADC shutdown test</td>
<td>This test verifies</td>
<td>-</td>
<td>Test</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>shutdown operation of</td>
<td>-</td>
<td>Group</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>ADC block</td>
<td>-</td>
<td>Port</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-</td>
<td>Cross</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Simplifying Generation of DO-254 Compliant Verification Documents for AEH Devices
by Hari Patel and Amarkumar Solanki, eInfochips

Figure 2: Illustrates an effective and fast testplan generation flow

STEP 1: CREATING A VERIFICATION CASE DOCUMENT (VCD)
Every requirement has to be verified in all possible scenarios. Therefore test scenario documents are very important in order to achieve 100% verification completeness. Since requirements of every project differ from each other, the test scenarios must be written for each project. An Excel sheet is a preferred format for documenting test scenarios, as it is easier to read the script in further processes. It can also be created in any tool which supports XLS file export capability. The avionics industry uses different tools to keep records of changes and document versions as per DO-254 standards.

As shown in the table below, the main fields of a test scenario document are Test Number, Test Name, and Test Scenarios. More columns can be added as required. This Excel sheet will be an input file for the script (discussed in Step 3), as show in Figure 2. Additional information related to tests can be added as this will be read by the script and will need no manual effort in merging.

Table 1: Sample of test scenario Excel document

<table>
<thead>
<tr>
<th>Test no.</th>
<th>Test Name</th>
<th>Test Description</th>
<th>Additional information</th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
<td>example_test.sv</td>
<td>&lt;Describe scenarios to be generated in the test&gt;</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>........</td>
<td>........................................</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>........</td>
<td>........................................</td>
<td></td>
</tr>
</tbody>
</table>

STEP 2: SCRIPT TO EXTRACT TESTBENCH AND COVERAGE INFORMATION AND CREATE VPD DOCUMENT
As per the DO-254 standard, an in-depth Verification Procedure Document (VPD) with detailed information is required to be created. Table 2 on the following page shows some examples of items that are present in VPD. The VPD is a a very lengthy document and has to be maintained to reflect changes in code. Creating and maintaining such a document manually, consumes a lot of time and is even prone to errors.

A VPD describes testbench, tests and coverage information, a Perl script can be created which can extract all coverage related information from the UCDB file created after running a test case or a regression. Users can maintain a certain style of the code while coding testbench and tests, so that the required information for tests, checkers, drivers, monitors, sequences, etc. can be easily extracted by the script. The coverage related information can be collected from the merged coverage file too. All the simulation tools provide facility for functional coverage extraction and generate the file in specific format. This file contains required information of cover groups, bins, weight, goal, assertions, etc. The Questa tool creates a UCDB file with all the above-mentioned information.
We can create a Perl script, which can take testbench code and a merged coverage file as an input, to extract different tests, testbench components and coverage related information. The output Excel file of this script is used as one of the inputs for the script described in Step 3. Refer to Figure 2 for the flow.

As shown in Table 2 above, the script should be smart enough to create detailed sheets with relevant information mentioned in columns including “Name”, “Purpose”, “Type” and “Source File”. The output excel sheet shall have detailed information about all verification components like driver, monitor, etc. It can also document checkers from monitor components. The script will automatically grab the description of each component mentioned in the “Name” column and puts it in the “Purpose” column. Information for “UCDB Link” and “Type”, related information can be mostly extracted from UCDB (coverage output) file. Information for “Source File” can be gathered through testbench code. Under “Trace” column, a trace ID has to be entered manually, which maps the test case of the VCD document with the particular VPD object.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Name</th>
<th>Purpose</th>
<th>Trace</th>
<th>UCDB Link</th>
<th>Type</th>
<th>Source File (Path)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>test_example</td>
<td>This test shows the example</td>
<td>106</td>
<td>example_test_link*</td>
<td>test</td>
<td>&lt;Path&gt;</td>
</tr>
<tr>
<td>2</td>
<td>example_monitor</td>
<td>This component explains monitor example</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>example_check</td>
<td>This check shows the example</td>
<td>106</td>
<td>cg_check_example</td>
<td>covergroup</td>
<td>&lt;Path&gt;</td>
</tr>
<tr>
<td>4</td>
<td>example_driver</td>
<td>This component explains monitor example</td>
<td></td>
<td></td>
<td></td>
<td>&lt;Path&gt;</td>
</tr>
<tr>
<td>5</td>
<td>example_coverpoint</td>
<td>This coverpoint contain example coverage</td>
<td>106</td>
<td>cg_example_block_c:cp_example</td>
<td>coverpoint</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Sample of VPD Excel file with coverage information

**SCRIPT DESCRIPTION**

We can create a Perl script, which can take testbench code and a merged coverage file as an input, to extract different tests, testbench components and coverage related information. The output Excel file of this script is used as one of the inputs for the script described in Step 3. Refer to Figure 2 for the flow.

As shown in Table 2 above, the script should be smart enough to create detailed sheets with relevant information mentioned in columns including “Name”, “Purpose”, “Type” and “Source File”. The output excel sheet shall have detailed information about all verification components like driver, monitor, etc. It can also document checkers from monitor components. The script will automatically grab the description of each component mentioned in the “Name” column and puts it in the “Purpose” column. Information for “UCDB Link” and “Type”, related information can be mostly extracted from UCDB (coverage output) file. Information for “Source File” can be gathered through testbench code. Under “Trace” column, a trace ID has to be entered manually, which maps the test case of the VCD document with the particular VPD object.

**TRACE ID**

In order to maintain traceability between output file VPD and test scenario document VCD, a unique link ID called “Trace” needs to be generated. This Trace ID can bridge a particular test scenario from one document to its coverage information mentioned in another document. Without adding Trace ID, the script mentioned in the final step will not be able to link scenarios from the first document to information provided in the second document. It is necessary to preserve the Trace ID when running the script again, or else manual effort needs to be put in for tracing scenarios and its output. To do that, the VPD document can be provided as an input to the script.

**STEP 3: SCRIPT TO CREATE TESTPLAN XLS**

One more Perl script can be created for final execution. This script takes both the sheets VCD and VPD created as an input and writes to a new Excel sheet to create testplan which will be an output of this script. This output Excel sheet will be a merged document of the information being read from the two input files.

Here, the role of the “Trace” column comes into the picture. The script will read test scenarios from the VCD document (created in Step 1) for a particular Trace ID and reads coverage information for the same Trace ID from
VPD (created in Step 2) and creates an output Excel sheet containing all the details, as shown in the table above.

This output Excel document will cover in-depth information of each test case and its coverage in a single document, which gives extra value to the verification plan. It serves as a proof of procedure that was followed for completely verifying each test case. Additional information as shown in Figure 1, like instance, weight and goal can be easily added through the script to this XLS sheet to create a final version of the XLS sheet required by the tool as input. Now this test plan Excel sheet can be saved as an XML file to convert to XML format and can be used by the simulation tool to generate final reports.

**Table 3: Sample of test plan Excel auto-generated by script**

<table>
<thead>
<tr>
<th>Trace</th>
<th>Test Name</th>
<th>Description</th>
<th>Link</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
<td>example_test.sv</td>
<td>&lt;…&gt;</td>
<td>example_test_link*</td>
<td>Test</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cg_chk_pass_cov, eg_a_check</td>
<td>covergroup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cg_chk_pass_cov, eg_b_check</td>
<td>covergroup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cg_example_cov.cp_example_a</td>
<td>coverpoint</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cg_example_cov.cp_example_b</td>
<td>cross</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cg_example_cov.cross_cover_example_a_b</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
</tbody>
</table>

**MERITS**

- One-time effort to create scripts that can be used in different verification projects
- Decreases human efforts
- Saves time and improves productivity
- Less errors while creating VPD documents
- Easy to maintain

**DEMERITS**

- Trace ID has to be filled correctly and manually
- Testbench must be written in specific style to maintain uniformity (works as an advantage too)
Late 2014, we found ourselves in a Project to develop a custom interconnect UVM Compliant VIP. Not only was there a need to develop a custom UVM VIP, but there was a need to plug this to a DUT which has a PCIe and an Avalon Streaming interface on it and perform the advance verification using our custom UVM VIP. The challenges were:

- Developing a custom interconnect UVM compliant VIP from scratch
- Verification of a DUT using the custom interconnect UVM VIP
- Verification of the PCIe interface of the DUT
- Making sure that the documentation created during the verification is DO-254 compliant
- Requirements, traceability information, functional coverage reports, code coverage reports need to be created

STEP 1: CREATING A PROJECT MANAGEMENT ENVIRONMENT

This verification project had to be compliant with DO-254. Therefore, not only did we have to plan the UVM environment and how we’re going to do the verification, we had to plan how we were going to manage different aspects of the project life cycle.

Requirements Management and Tracing:

We used conventional methods here, Word and Excel. The requirements are captured in MS Word, the verification plan is developed again in MS Word and requirements traceability matrix is developed in MS Excel. And this is REAAALY cumbersome. Why? Here are the reasons why we’re going to switch to a professional requirements management and tracing system in our next project.

Requirements Review is hard with MS Word. Reviewers were using the “Track Changes” feature of MS Word and each review was being committed to the revision control system. Tracking when the review is done, who has done the review is all manual. Also it’s not possible to generate queries related to reviews. Requirement numbering was also manual which is cumbersome when new requirements are to be defined. A requirement management tool will be a better choice and make our life easier in terms of review logging and tracking. It will also be possible to use queries to extract different kinds of information related to requirements with a requirements management tool.

Revision Control:

SVN is used for revision control. It’s free and does the work for revision control. We used it for it’s tight integration to Trac and also our configuration management environment is based on SVN.

Configuration Management:

Configuration management is very important in a DO-254 compliant project. An internal configuration management tool is used based on SVN. This allowed us to create configuration items, generate SVN tags specific to configuration items, build configurations and embed configuration items in configurations. This method allowed us to run regressions on tagged top-level configurations and track which configuration item is used inside a top level configuration. The verification environment is reproducible and reverification, even after months for a specific top level configuration which has been released, is possible this way. Release notes for the configuration are also being generated automatically depending on which files are modified on SVN. Engineers can easily generate release note data without having to remember which changes to which files are done and our internal configuration management tool is providing a list of modifications before tagging a configuration item.

Project/Change/Defect Management:

There needs to be a Change/Defect Management system deployed and used throughout the project, especially in a DO-254 compliant hardware development project. We used Trac for Defect and Change Management. The revision control system was SVN.

Trac is a great (and free) software which can interface to Subversion, Git and other version control systems. Trac allows wiki markup in issue descriptions and commit messages, creating links and seamless references between bugs, tasks, change sets, files and wiki pages. A timeline shows all current and past project events in order, making the acquisition of an overview of the project and tracking...
progress very easy. The roadmap shows the road ahead, listing the upcoming milestones. See http://trac.edgewall.org/ for further information.

**STEP 2: GENERATE THE REQUIREMENTS AND THE PROJECT ENVIRONMENT**

After two months of project kick-off we had the first version of the requirements. Then, we started to develop the UVM environment with the initial requirements at hand. However, the requirements were not frozen. Have you seen any project where the requirements are frozen throughout the project? Probably not. And our case was not an exception. Therefore, requirements are changed and changed and modified and some of them removed and again changed, modified, and so on...... This is why managing and tracking requirements using MS Word and MS Excel is not a good idea as when a requirement is changed/modified/deleted, we need to also modify the verification plan and requirements traceability matrix which are not linked. This modification process is error prone and more than this, time consuming. Lessons learned. It’s time for us to use a requirements management tool and a traceability tool like ReqTracer™ in future projects.

**STEP 3: PLAN & DEVELOP UVM ENVIRONMENT**

The initial work was to generate a top level block diagram of the UVM environment and come to a common agreement within the team on what will be coded, which UVM items, sequences, monitors, drivers, agents to be developed; create a plan and estimate the work load for the entire verification development. The micro planning has been done, milestones are identified and milestones are entered into Trac. Knowing our milestones is crucial as your defects, change requests, tasks with owners, priorities and related configuration items will all be linked to milestones. We had to know which item we were entering into Trac would be for which milestone. This way weekly progress of the project could be tracked for every item entered into the Trac system. Project plan was also aligned to the items in the Trac system and that facilitated the project management.

For the custom interconnect UVM VIP, the initial work was to develop the UVM environment base classes for the agent, sequence item, monitor, sequencer, environment configuration. This took almost one month with two people.

Another engineer tried to integrate Mentor’s PCIe VIP to the DUT and run the initial tests. This took a couple of days to have the verification environment up and ready, thanks to Mentor’s Altera Kit to integrate PCIe QVIP. Mentor provided an example kit to integrate their PCIe VIP into Altera’s PCIe Hard IP and this made our life easier during the first bring up of the PCIe interface; we saw that there was a link established and a bus enumeration could be done. The documentation was also good quality and most of the time we were able to find the answer to a question related to PCIe VIP in the documentation without contacting Mentor Support.

**STEP 4: TEST CASE IMPLEMENTATION & REGRESSION & RESULT ANALYSIS**

We had close to 200 test cases to be implemented. All of them were random test cases. They were linked to the requirements. For each test case, we defined the assertions, cover directives coverpoints and covergroups for our DUT. Questa® Prime was used as the HDL simulator, we gathered them in a test plan in XML format to be able to create a test_plan.ucdb. Creating a test_plan.ucdb allowed us to link test cases defined in our Verification Plan to the Regression Results UCDB file. This way, after running a regression, it was possible to see the total coverage (code or functional) of the regression and also for each test case. It was also possible to analyze which cover items were not covered in which test case and this made verification engineers’ coverage analysis much easier and faster.

We initially created a regression suite based on C-shell scripts. We created a C-shell script which could run a single test case or run a regression by read in a test file where all the test case names were defined. Script can accept some command line arguments like GUI mode, test case name, UVM verbosity level, waveform dump file, coverage on/off, seed, etc. The result analysis was done at the end of each test case run by checking some predefined terms like “Error: “UVM_ERROR”, “UVM_FATAL", etc. There were two drawbacks of this regression C-shell script:
1. Running test cases in parallel to speed up the regression time was not possible
2. Running the tests with different repeat numbers was not supported with the script. This was needed to be able to increase functional coverage by repeating the run times for the test cases

As a result of the needs listed above, we did a pilot evaluation of Verification Run Manager (VRM) tool from Mentor. The deployment was easy, thanks to the support from Mentor. Once the regression environment was up and running using VRM, our regression times improved (of course you're limited here with the number of Questa licenses you have as it is a parallel run) and we had a very nice way to run the test cases in a regression with different seeds, different repeat numbers and different run time options. VRM is also working in an integrated way with Questa hence, we were able to debug a failing test in an interactive way as a result of the regression run. Another benefit was “Result Analysis”. VRM can capture UVM errors, fatals, warnings, infos for each test case run and generate a nice report for further debugging and project reporting. What can be captured can also be customized, meaning we can search for a specific word/sentence in the results of test cases and place them in a result analysis group for further investigation.

**NEXT STEPS**

Developing a UVM environment, especially for VIP, takes time, requires experience. However, once it is there, with the right tools in hand, the confidence of the verification environment is definitely increased. Because we now know that the methods and the environment itself can be reused in our next projects and because it is UVM, when a new engineer with UVM knowledge starts with our team, he/she will already be equipped with the necessary information to get up and running with us. There is no need to teach him/her company internal methods/tools/methodologies for verification.

We now have ReqTracer, Register Assistant and Questa inFact in our “To Do list” to be learned and to be deployed.
INTRODUCTION

Testbenches written in SystemVerilog and UVM face the problem of configurability and reusability between block- and system-level. Whereas reuse of UVCs from a block- to a system-level verification environment is relatively easy, the same cannot be said for the UVC’s connection to the harness: The interfaces that these UVCs need changes from connections to primary inputs and outputs at block level to a set of hierarchical probes into the DUT at system level. This requires a re-write of all interface connections and hinders reuse.

This article demonstrates how to write interface connections only once and use them in both block- and system-level testbenches. The order is not important: System-level testbenches can be written without all the blocks of the DUT completed, DUT and UVM blocks can easily be interchanged. Taking care not to use virtual interfaces in the UVC but Bus Functional Models (BFM) in the interface instead – so called polymorphic interfaces, UVCs can be fully configurable as well as reusable.

REVIEW OF INTERFACES, THEIR REUSABILITY AND CONFIGURABILITY AND LIMITATIONS

During the development of a chip, it is usual to write blocks and quite often verify them stand-alone. These blocks would then typically be grouped into larger blocks / subsystems before a number of them get integrated into the complete chip. This means that the testbench needs to be scaled as the DUT grows. Modern testbenches often make use of UVM’s class based verification environment. This focuses very much on reuse, mainly vertically from block level up to chip level, but also horizontally between blocks. The weak link, however, is on the one hand the connection between the static module hierarchy of the DUT (which is typically instantiated inside the testbench) and on the other hand the dynamic class-based verification environment.

Like modules, an interface is created at elaboration time, and therefore testbench writers would instantiate all interfaces which are required for a DUT in the testbench and connect them to the DUT, which is also instantiated in the testbench. Although SV interfaces were (also) meant to be a “synthesizable collection of signals” to ease the onerous connection of DUT blocks, this approach has not yet been widely adopted. This approach results in an equally onerous task of connecting the DUT in the testbench: Each signal has to be connected individually. Worse, these connections cannot be reused vertically: When the block is integrated into a subsystem, it is often desirable to keep the connection between the agent and the block and switch the agent into passive mode. This requires that the interface is connected to signals which are now buried inside the subsystem. This is not only tedious, but it may be that in post-synthesis these signals are difficult to find.

From the testbench, the SV interface is then passed into the verification environment. This could be done by either a set_vi() access function, by writing a wrapper or configuration class around the interface and passing it into the agent via UVM’s configuration mechanism or, since the advent of the uvm_config_db, by passing the virtual interface directly into the uvm_config_db. This has the disadvantage that it hinders horizontal reuse, because the virtual interface type includes the type specialization, so any parameters given to the interface in the testbench must be known in the agent at compile time. Typical code would look like this:

```vhs
interface bus_if #(int ADDR_W = 16, DATA_W = 16)
    (input logic clk,
     input logic rst_n);
    logic [ADDR_W-1:0] addr;
    logic [DATA_W-1:0] data;
 //...
endinterface: bus_if

module block #(int AW = 16, DW = 16)
    (input logic clk,
     input logic rst_n,
     input logic [AW-1:0] addr,
     input logic [DW-1:0] data, ...);
 ...
endmodule: block

class bus_agent #(int AW = 16, DW = 16)
extends uvm_component;
virtual interface bus_if #(ADDR_WIDTH (AW),
     .DATA_WIDTH (DW)) vif;
 //...
endclass: bus_agent
```
Another problem is that the agent needs to know about the virtual interface's specialization. This information can only come from the agent's own specialization, from a define command or from a package containing all configuration options unambiguously.

WRITING A REUSABLE VERIFICATION FRAMEWORK

The problem of the interface connections can be solved by changing the place where the interface is instantiated. Using the bind command to bind it into the module, the compiler treats it as if it was inside the module, giving it access to all the module's signals. By changing the interface declaration such that it uses ports instead of internal signals for the connections to the DUT, we can write the port list in the bind command in such a way that it connects to any signal in the DUT:

```verilog
module tb;
  bit clk, rst_n;
  bus_if #(ADDR_W (16), DATA_W (16)) bus_if_inst (.clk (clk), .rst_n (rst_n));
  block #(AW (16), DW (16)) DUT (.clk (clk), .rst_n (rst_n), .addr (bus_if_inst.addr), .data (bus_if_inst.data), ...);

  uvm_config_db# (virtual interface bus_if # (16, 16))::set (null, "...", "vif", bus_if_inst);

endmodule: tb
```

The bind command instructs the compiler to add the instantiation `bus_if_inst` of type `bus_if` to the module `small_block`. It binds it to the type `small_block`, so each instance of `small_block` will have an instance of the interface. So the signals `bus1_addr`, etc., are signals in `small_block`. It is also possible to bind the interface to a specific instance of the module only. Note that the type specializations of the module and the interface need to match, otherwise the signals which are being connected are of different width. In this instance, we want to connect the interface only to the ports of the DUT, as described in [1]. It is also recommended in [2] to use this approach for probing into the DUT, where a probe interface is bound into the DUT and connected to any internal signal. [2] also shows how to use clocking blocks and modports, which are omitted here.

Note that the direction of the port in the interface cannot be output. An output port adds a driver to the signal, which prohibits reuse of the interface in passive mode. The bind command has the same pitfalls as a "real" instantiation, like declaring signals implicitly when a connection is mistyped.

The problem of the type specialization in the agent can be avoided by eliminating the need for the agent to have a virtual interface in the first place. This can be done by using bus functional models, where a virtual base class for the BFM containing function prototypes is written which extends from uvm_component. This base class is used inside the agent and is extended in the interface, where the function prototypes are implemented. If a wrapper class is used in the same way providing a build function for the BFM, the building of the BFM can be done in the correct UVM phase and thus make it configurable via the usual UVM configuration mechanisms in the same way as any other uvm_component is configurable. [3] A function `get_api_wrapper()` must also be implemented in the interface, building an instance of the wrapper class and returning a handle on it.

```verilog
interface bus_if #(int ADDR_WIDTH = 16, int DATA_WIDTH = 16)
  (input logic clk,
   input logic [ADDR_WIDTH-1:0] addr,
   input logic [DATA_WIDTH-1:0] wdata,
   input logic [DATA_WIDTH-1:0] rdata,
   bind small_block bus_if #((int ADDR_WIDTH = 16),
   DATA_WIDTH (16)) bus_if_inst
  (.clk (clk),
   .rst_n (rst_n),
   .addr (bus1_addr),
   .wdata (bus1_wdata),
   .rdata (bus1_rdata),
   // ...
  );

The bind command instructs the compiler to add the instantiation `bus_if_inst` of type `bus_if` to the module `small_block`. It binds it to the type `small_block`, so each instance of `small_block` will have an instance of the interface. So the signals `bus1_addr`, etc., are signals in `small_block`. It is also possible to bind the interface to a specific instance of the module only. Note that the type specializations of the module and the interface need to match, otherwise the signals which are being connected are of different width. In this instance, we want to connect the interface only to the ports of the DUT, as described in [1]. It is also recommended in [2] to use this approach for probing into the DUT, where a probe interface is bound into the DUT and connected to any internal signal. [2] also shows how to use clocking blocks and modports, which are omitted here.

Note that the direction of the port in the interface cannot be output. An output port adds a driver to the signal, which prohibits reuse of the interface in passive mode. The bind command has the same pitfalls as a "real" instantiation, like declaring signals implicitly when a connection is mistyped.

The problem of the type specialization in the agent can be avoided by eliminating the need for the agent to have a virtual interface in the first place. This can be done by using bus functional models, where a virtual base class for the BFM containing function prototypes is written which extends from uvm_component. This base class is used inside the agent and is extended in the interface, where the function prototypes are implemented. If a wrapper class is used in the same way providing a build function for the BFM, the building of the BFM can be done in the correct UVM phase and thus make it configurable via the usual UVM configuration mechanisms in the same way as any other uvm_component is configurable. [3] A function `get_api_wrapper()` must also be implemented in the interface, building an instance of the wrapper class and returning a handle on it.

```
The testbench can then build the wrapper and get a handle on it by simply calling the interface’s `get_api_wrapper()`.

Using Bus Functional Models also has the advantage that it enables acceleration through emulation. For that, the (timed) DUT, which runs on an emulator, needs to be strictly separated from the (untimed) verification code, which runs on a simulator. They are only allowed to communicate via transactions. [4]

Let us look at an example of a small block with a simple bus slave interface. To reuse this block both horizontally and vertically, we integrate it into a big block, together with a control block and memory. To keep it simple, the big block uses the same bus slave interface, which is passed into the control module. This uses the most significant address bit to direct the transaction to either the small block or the memory, as shown below.

This is the log when the DUT is the small block. It shows that we have an agent with a BFM configured as active, driving some data into the DUT, where it is reported when the transaction arrives. For the subsystem framework, we want to keep this agent and interface, but put it into passive mode, so that it only monitors the signals and reports ongoing activity. We need another interface and agent of the same type but with different address and data widths to connect to the bus in `big_block`. There is a deliberate data width mismatch in the DUT to demonstrate the use of different type specializations within the same framework (See table on the following page).

Looking at this log, we can see that we have configured the agent connecting to `small_block` into passive mode and the one connecting to `big_block` into active mode. We can further see in the log that a data transaction initiated by the BFM is seen by the monitor BUSMON in the now passive interface, before it is reported in the `small_block`.

**CONCLUSION**

Using Bus Functional Models is certainly a good way to partition a verification environment. If the BFM needs to be configurable, care must be taken to build it in the correct phase in the UVM build process and thus logically insert the BFM into UVM’s hierarchy. It also enables emulation if the BFM’s access functions are written according to the emulator’s requirements. Using the bind command to instantiate interfaces into the DUT allows us to connect a block’s ports once and reuse those connections at subsystem or chip level. Following these practices allows both horizontal and vertical reuse.

---

```
virtual class virtual_bfm extends uvm_component;
virtual_active_passive_enum m_active;
// define API here, anything the agent needs to have access to or from
pure virtual task wr_packet (uvm_bitstream_t l_addr, uvm_bitstream_t l_data);
pure virtual task rd_packet (uvm_bitstream_t l_addr, ref uvm_bitstream_t l_data);
pure virtual function void some_api();
// …
endclass: virtual_bfm

interface bus_if...
class concrete_bfm extends virtual_bfm ...
function concrete_bfm_wrapper get_api_wrapper ...
endinterface
```

The Name | Type          | Size | Value
---------|---------------|------|-----------------------------
| uvm_test_top | my_test      | -    | @2611
| m_bus_agent  | bus_agent    | -    | @2709
| m_bfm        | concrete_bfm | -    | @2655
| m_active     | uvm_active_passive_enum | 1 | UVM_ACTIVE
| m_bfm_wrappe  | concrete_bfm_wrapper | - | @2649
| m_active     | uvm_active_passive_enum | 1 | UVM_ACTIVE

---

UVM_INFO @ 15: uvm_test_top.m_bus_agent.m_bfm [some_api] m_active: 1, AW=16, DW=16
UVM_INFO @ 80: uvm_test_top.m_bus_agent.m_bfm [wr_packet] addr=1234, data=5678
UVM_INFO @ 80: reporter [small_block] BUS1 write 5678 to addr 1234
<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uvm_test_top</td>
<td>my_tes</td>
<td>-</td>
<td>@2614</td>
</tr>
<tr>
<td>m_agent_big_mod</td>
<td>bus_agent</td>
<td>-</td>
<td>@2767</td>
</tr>
<tr>
<td>m_bfm</td>
<td>concrete_bfm</td>
<td>-</td>
<td>@2661</td>
</tr>
<tr>
<td>m_active</td>
<td>uvm_active_passive_enum</td>
<td>1</td>
<td>UVM_ACTIVE</td>
</tr>
<tr>
<td>m_bfm_wrapper</td>
<td>concrete_bfm_wrapper</td>
<td>-</td>
<td>@2658</td>
</tr>
<tr>
<td>m_active</td>
<td>uvm_active_passive_enum</td>
<td>1</td>
<td>UVM_ACTIVE</td>
</tr>
<tr>
<td>m_agent_sml_mod</td>
<td>bus_agent</td>
<td>-</td>
<td>@2726</td>
</tr>
<tr>
<td>m_bfm</td>
<td>concrete_bfm</td>
<td>-</td>
<td>@2856</td>
</tr>
<tr>
<td>m_active</td>
<td>uvm_active_passive_enum</td>
<td>1</td>
<td>UVM_PASSIVE</td>
</tr>
<tr>
<td>m_bfm_wrapper</td>
<td>concrete_bfm_wrapper</td>
<td>-</td>
<td>@2653</td>
</tr>
<tr>
<td>m_active</td>
<td>uvm_active_passive_enum</td>
<td>1</td>
<td>UVM_PASSIVE</td>
</tr>
</tbody>
</table>

UVM_INFO @ 10: uvm_test_top.m_agent_sm.m_bfm [some_api] m_active: 0, AW=19, DW=16
UVM_INFO @ 10: uvm_test_top.m_agent_bm.m_bfm [some_api] m_active: 1, AW=20, DW=32
UVM_INFO @ 60: uvm_test_top.m_agent_bm.m_bfm [wr_packet] addr=12345, data=56787654
UVM_INFO @ 60: uvm_test_top.m_agent_sm.m_bfm [BUSMON] write 7654 to addr 12345
UVM_INFO @ 60: reporter [small_block] BUS1 write 7654 to addr 12345
UVM_INFO @ 120: uvm_test_top.m_agent_bm.m_bfm [wr_packet] addr=87654, data=deadbeef
UVM_INFO @ 140: reporter [mem] wrote deadbeef to addr 07654

REFERENCE

[1] A proven methodology to hierarchically reuse interface connections from the block to the chip level,
   David Larson
[2] The Missing Link: The Testbench to DUT Connection,
   David Rich, 2012
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

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

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 6500 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com