I must confess that I’m not very handy around the house. Oh sure, I know how to do more than just change a light bulb, and I’ve tackled the occasional clogged drain or recalcitrant electrical outlet, but I know my limits. This is one of the reasons that my wife and I decided to build a new house rather than buy an old one. It was much easier to tell the builders where to put the wall before it went up than it would have been for me to move a wall that was already there.

Besides, there was something comforting in being there when they actually poured the foundation for our house and in watching it take shape where I could be confident that it was a strong and sturdy house. And the fact that I knew I wouldn’t have to be doing any major repairs to anything made what most people find a very stressful process anything but.

I daresay that my wife was equally at ease (perhaps more so) knowing that my carpentry skills wouldn’t need to be stretched, either.

I was reflecting on this idea of building on a strong foundation recently, and it occurred to me that relying on good solid work that others have done is often the key to productivity...since it lets us focus on what we need to and lets us avoid wasting time and resources...

---

A Strong Foundation Is Always a Good Start.
By Tom Fitzpatrick, Editor and Verification Technologist

I must confess that I’m not very handy around the house. Oh sure, I know how to do more than just change a light bulb, and I’ve tackled the occasional clogged drain or recalcitrant electrical outlet, but I know my limits. This is one of the reasons that my wife and I decided to build a new house rather than buy an old one. It was much easier to tell the builders where to put the wall before it went up than it would have been for me to move a wall that was already there.

Besides, there was something comforting in being there when they actually poured the foundation for our house and in watching it take shape where I could be confident that it was a strong and sturdy house. And the fact that I knew I wouldn’t have to be doing any major repairs to anything made what most people find a very stressful process anything but. I daresay that my wife was equally at ease (perhaps more so) knowing that my carpentry skills wouldn’t need to be stretched, either.

I was reflecting on this idea of building on a strong foundation recently, and it occurred to me that relying on good solid work that others have done is often the key to productivity...since it lets us focus on what we need to and lets us avoid wasting time and resources...

---

“...it occurred to me that relying on good solid work that others have done is often the key to productivity...since it lets us focus on what we need to and lets us avoid wasting time and resources...”

—Tom Fitzpatrick

Our lead article in this issue once again comes from our Assertions Guru, Harry Foster. As FPGAs are becoming more sophisticated and more popular, traditional FPGA flows are simply not able to keep up with the growing verification complexity. Harry shows how the addition of assertion-based verification (ABV) can evolve your existing process both to improve the quality of your designs and to reduce the amount of time spent debugging problems when they occur.

As Harry mentions, the adoption of ABV in your simulation flow then opens the door for the use of static and dynamic formal verification as well. Our next article comes from Ping Yeung, of our Verification Technology team, who introduces you to the wide spectrum of advanced static verification. The foundation for this approach is the automatic analysis of the design
(and assertions, if there are any) to detect any common problems early in the design cycle, before you have a completed testbench or even before the design is complete. Everything from lint checking for common coding problems to autogeneration of assertions to check clock-domain crossing errors can be used to avoid many problems that often don’t show up until too-late in a traditional simulation-based flow.

The use of constrained-random stimulus in a coverage-driven verification environment has certainly come to be a foundational concept for functional verification. Relying on randomly-generated variables subject to constraints to ensure their validity has been proven to uncover those pesky “corner-case” bugs that invariably crop up when large systems are assembled from many different sources. The value comes from the unpredictability of the stimulus allowing your testbench to exercise scenarios you might not have thought about. The problem with constrained-random stimulus, though, is that often the random stimuli end up producing redundant patterns (wasting simulation cycles) or sometimes not quite reaching a particular coverage point. In our next article, Matt Ballance of our inFact team shows how inFact’s coverage-driven stimulus generator can analyze both the stimulus and coverage models and produce a much more efficient stimulus set that can achieve your coverage goals without redundancy while still maintaining the value of random generation.

Relying on the work of others allows us to avoid “reinventing the wheel.” In verification, the most obvious application of this concept is the use of pre-packaged verification IP, such as Questa MVCs. In “Verification of an Ethernet PHY DUT using Questa MVCs,” Pankaj Goel provides a strategic view of how to take advantage of the configurable features of an MVC to handle many common verification problems associated with the ethernet protocol.

Just as a builder relies on other contractors to build the house, we are pleased and proud of our Questa Vanguard Partners who help to round out, not only our product and service offerings, but Verification Horizons itself. In our Partners’ Corner section of this issue, we begin with our friends from Silicon Interfaces, who provide “a Design Engineer’s Perspective on Verification Using Questa and OVM.” The title pretty much says it all. Next, the folks at Paradigm Works discuss their new scoreboard component, part of their SystemVerilog Frameworks for OVM. Third, we have a discussion by Gleichmann Electronics Research about their SEmulator®, which bridges the gap between simulation and FPGA prototyping, allowing you to move design blocks between the two domains. And last but not least for our partners, our long-time friend Cliff Cummings of Sunburst Design shows you how to write code concise SystemVerilog Assertions.

As many of you know, Cliff is a master of Verilog and SystemVerilog coding tricks, and this article serves as another “rabbit” in his magic hat.

We close this issue with a user article on a “Methodology for Board Level Functional Simulation and Hardware/Software Co-Verification Using Seamless” from Alcatel-Lucent. This article provides a practical case study of how to use Seamless to accomplish early hardware/software integration. It also discusses the advantages they saw from investing in this technology and how it will provide them a solid foundation on which to build in the future.

A good foundation can make all the difference. Whether you’re building a house or verifying an electronic system, a strong foundation means less for you to worry about. For me, it’s easier to build a testbench than it is to build a cabinet, but that’s partly because my carpentry skills were learned building sets for community theater (where I met my wife). In theater, the key rule is that no one in the audience will ever get closer than 30 feet from the results. Of course, you can’t afford such an approach to verification, so I hope that this issue of Verification Horizons will provide you a strong foundation on which to build.

We are also excited to share with our readers the latest expansion of Verification Horizons. You no longer have to wait for your latest issue to hear from our authors, as you can now hear from our team, including Harry Foster, Dennis Brophy and others, weekly online in the NEW Verification Horizons BLOG at VERIFICATIONHORIZONSBLOG.COM. We hope you’ll find this a useful tool in building your Verification Foundation.

Respectfully submitted,

Tom Fitzpatrick
Verification Technologist
Mentor Graphics
“...you can now hear from our team, including Harry Foster, Dennis Brophy and others, weekly online in the NEW Verification Horizons BLOG at VERIFICATION HORIZONSBLOG.COM.”
# TABLE OF CONTENTS

*Page 6... Evolving Your Organization’s ABV Capabilities*
by Harry Foster, Chief Verification Scientist, DVT Mentor Graphics

*Page 10... Advanced Static Verification is Indispensable*
by Ping Yeung Ph.D., DVT Mentor Graphics

*Page 14... Evolving the Coverage-Driven Verification Flow*
by Matthew Ballance, Technical Marketing, SLE Division, Mentor Graphics

*Page 17... Verification of an Ethernet PHY DUT Using Questa MVC*
by Pankaj Goel, Mentor Graphics

**Partners’ Corner:**

*Page 22... Breaking Your Own Code; A Design Engineer’s Perspective on Verification Using Questa and OVM.*
by Leeja Mathew, Mahasweta Das, Reshma Shetty; Silicon Interfaces
Page 26... SystemVerilog FrameWorks™ Scoreboard: An Open Source Implementation Using OVM.
by Dr. Ambar Sarkar, Chief Verification Technologist, Paradigm Works Inc.

Page 31... SEmulation: Use your emulation board as a hardware accelerator for ModelSim SE.
by Andreas Schwarztrauber, Gleichmann & Co. Electronics GmbH

Page 35... Coding Concise SystemVerilog Assertions.
by Clifford E. Cummings, Sunburst Design, Inc.

Page 41... Methodology for Board Level Functional Simulation and Hardware/Software Co-Verification Using Seamless.
by John Gryba, Senior Hardware Design Engineer, Alcatel-Lucent
Ensuring functional correctness on RTL designs continues to pose one of the greatest challenges for today’s FPGA and ASIC design teams. Very few project managers would disagree with this statement. In fact, an often cited 2004 industry study by Collett International Research revealed that 35 percent of the total ASIC development effort was spent in verification. [1] In 2007, a Far West Research study indicated the verification effort has risen to 46 percent of the total ASIC development effort. [2] Furthermore, these industry studies reveal that debugging is the fastest-growing component of verification, and that it consumes 52 percent of the total verification effort. Unfortunately, with the increase in verification effort, the industry has not experienced a measurable increase in quality of results. For example, the Far West Research study indicated that only 28 percent of projects developing ASICs were able to achieve first silicon success. To make matters worse, the industry is witnessing increasing pressure to shorten the overall project development cycle. Clearly, new design and verification techniques, combined with a focus on maturing functional verification process capabilities within an organization (and the industry as a whole) are required.

Today we are witnessing a phenomenal increase in FPGA design starts as one means to reduce risk. In fact, Gartner recently reported that FPGAs now have a 30-to-1 edge over ASICs in design starts. Although FPGAs have traditionally been relegated to glue logic, low-volume production, or prototype parts used for analysis, this is no longer the case. Gate count and advanced features found in today’s FPGAs have increased dramatically to compete with capabilities traditionally offered by ASICs alone. The change in FPGA capabilities has results in the emergence of advanced FPGA system-on-chip (SoC) solutions, which includes the integration of third-party IP, DSPs, and multiple microprocessors—all connected through advanced, high-speed bus protocols. Accompanying these changes has been an increase in design and verification complexity, which traditional FPGA flows are generally not prepared to address. In this article, I talk about an easy technique for addressing verification complexity by evolving your organization’s simulation process capabilities—specifically through the adoption of assertion-based verification (ABV).

WHAT IS AN ASSERTION?

Informally, an assertion is a statement of design intent that can be used to specify design behavior. Assertions may specify internal design implementation features. For example, some aspect of a specific FIFO structure. Alternatively assertions may specify external architectural (or micro-architectural) features. For example, a bus protocol or even higher-level, end-to-end behavior that spans multiple design blocks.

One key characteristic of assertions is that they allow us to specify what the design is supposed to do at a high level of abstraction, without having to describe the details of how the design was implemented. Thus, this abstract view of the design intent is ideal for the verification process—whether we are specifying high-level requirements or lower-level implementation behaviors—by means of assertions.

Let’s examine a simple implementation assertion. For this example we use a simple two-client arbiter, as illustrated in Figure 1.

![Figure 1. Simple Two-Client Arbiter](image)

A requirement of our simple two-client arbiter is that the grants remain mutually exclusive to prevent resource conflicts when accessing a shared resource. In other words, for our specific example, the arbiter should never assert gnt[0] and gnt[1] at the same time. We can specify this requirement using a SystemVerilog assertion, as demonstrated in Figure 2.

```
assert property ( @(posedge clk) disable iff (rst) ! (gnt[0] & gnt[1]));
```

![Figure 2. SystemVerilog Assertion For Mutually Exclusive Grants](image)

DOES THIS STUFF REALLY WORK?

Now the benefit of having this particular assertion present during the course of simulation, is that if there is a bug in our arbiter...
implementation, which caused two grants to be asserted at the same time, then our assertion would trigger. Thus, any improper or unexpected behavior can be caught closer to the source of the design error, in terms of both time and location. Hence, the use of assertions dramatically reduces our debugging effort.

When asked what the biggest bottleneck in their verification flows, the Far West Research participants responded that debugging was their issue, as illustrated in Figure 3. Hence, anything that can be done to reduce the debugging effort is a huge win for the overall project schedule.

When assertions significantly reduced debugging time, as measured and described in the following case studies:

- 50 percent reduction in simulation debugging time published by IBM [3]
- 50 percent reduction in simulation debugging and 85 percent reduction in formal debugging time published by Sun Microsystems [4]

From these case studies, a common theme emerges: when design and verification engineers use assertions as a part of the methodology, they are able to detect a significant percentage of design failures while reducing debugging time as a result of improved observability.

**CONTROLLABILITY AND OBSERVABILITY**

Fundamental to the discussion of assertion-based verification is understanding the concepts of controllability and observability. [5] Informally, controllability refers to the ability to influence or activate a specific line of code within the design by stimulating various input ports. Note that, while in theory a simulation testbench has high controllability of the design model’s input ports during verification, it can have very low controllability of an internal structure within the model. Observability, in contrast, refers to the ability to observe the effects of a specific stimulated line of code. Thus, a testbench generally has limited observability if it only observes the external ports of the design model (because the internal signals and structures are often indirectly hidden from the testbench).

To identify a design error using a simulation testbench approach, the following conditions must hold:

1. The testbench must generate proper input stimulus to activate a design error.
2. The testbench must generate proper input stimulus to propagate all effects resulting from the design error to an output port.

It is possible, however, to set up a condition where the input stimulus activates a design error that does not propagate to an observable output port, as illustrated in Figure 4.

Embedding assertions in the design model increases observability. In this way, the testbench no longer depends on generating input stimulus to propagate a design error to an observable port. In fact, one observation made by teams just getting started with ABV, is that after the assertions are enabled, their simulations that had previously been working often fail as new bugs are identified due to improved observability. Thus resulting in an overall reduction of debugging time.

While embedded assertions help solve the observability challenge in simulation, they do not help with the controllability challenge. However, the existence of assertions within the flow does open up the possibility for utilizing formal property checking to target critical or high-value assertions, thus addressing the controllability challenge.
WHO SHOULD WRITE THE ASSERTIONS?

A question I often hear by engineers just getting started with ABV is “who should write the assertions?” Assertions added at any level of hierarchy clearly benefit verification by reducing debugging time while clarifying design intent. Certainly multiple stakeholders within the design and verification process can contribute to the assertion development process—thus reducing ambiguities while improving observability.

Figure 5 illustrates a typical design refinement process through various levels of abstraction and the stakeholders associated with each level. Adoption of assertions in the industry, at the time of this writing, has predominately occurred within the block and module level. They can be called implementation assertions. This adoption trend is partially due to the lack of effective guidelines for assertion use at higher levels of design hierarchy (or abstraction) and confusion about which stakeholders should contribute to the assertion development process.

Although an architect can contribute to the assertion development effort by defining global properties (derived from the architecture and micro-architectural specification) that must hold across multiple possible implementations, the design engineer contributes by writing internal white-box assertions derived from the implementation. In addition, the verification engineer contributes by developing assertions that specify correct interface behavior between units and between blocks. The verification engineer also contributes by developing black-box, end-to-end assertions across design components.

From a verification planning perspective, the verification engineer generally does not track any of the low-level implementation assertions added to the RTL by the design engineer. These low-level assertions are analogous to checking that a pointer is not NULL before it is used in a software program. They are invaluable at isolating problems, and you will be very glad they are there when a problem occurs (to simply debugging), yet these assertions do not map back to any requirements defined in the architectural or micro-architectural specification. For the mature design team, who have applied an ABV methodology on multiple projects, you will typically find tens of thousands of these low-level assertions for larger designs (for example, >10M gates).

The verification engineer is generally concerned with assertions above the RTL implementation (that is, architectural and micro-architectural assertions. Examples of these type of assertions include block-level interfaces and bus protocols, as well as multi-block end-to-end assertions. Unlike low-level implementation assertions, these higher-level assertions are often defined and tracked as part of the verification planning process.

GETTING STARTED WITH ABV

There are two industry assertions language standards available today:

- An assertion language embedded in the IEEE 1800-2005 SystemVerilog standard known as SVA
- The IEEE 1850-2005 Property Specification Language PSL

In addition, there is an assertion library standard from Accellera known as the Open Verification Library (OVL) [6], which is available in Verilog, VHDL, SystemVerilog, and PSL.

From the perspective of an FPGA or ASIC designer who is just getting started with assertions, the OVL provides a simple learning curve for adoption. The designer can instantiate one of the Verilog or
VHDL OVL checkers directly into their RTL design without having to learn an assertion language. In fact, what we see today is that often these low-level implementation assertions are just simple Boolean checks, which do not require the full power of either PSL or SVA.

From a verification engineer’s perspective, PSL or SVA serves their needs better due to the expressiveness of these languages. These engineers are focused on writing higher-level assertions, such as bus protocols or end-to-end assertions involving multiple blocks.

NEXT STEPS

To learn more about ABV and how to evolve your organization’s verification process capabilities, visit the Verification Academy web site at www.verificationacademy.com.

Also, we are actively working on our next module for the Verification Academy, to be titled FPGA Verification. The new module is organized into eight sessions and is based on Ray Salemi’s book titled FPGA Simulation. [7] This module provides a step-by-step guide to evolving an FPGA project team’s verification capabilities and covers topics such as: code coverage, test planning, assertions in the FPGA world, transactions, self-checking test-benches, automatic stimulus, and functional coverage. Although the primary target for this module is project teams in the FPGA space, this module also provides an excellent basic introduction to advanced functional verification techniques. The first set of sessions are now available.

REFERENCES


[2] FarWest Research 2007 industry study in conjunction with Mentor Graphics,


Advanced Static Verification is Indispensable

by Ping Yeung Ph.D., DVT Mentor Graphics Corporation

I. INTRODUCTION

Traditionally, simulation-based dynamic verification techniques — such as directed tests, constrained-random simulation, and hardware acceleration — have been the work horse of functional verification. As modern day SoC designs become more integrated, the only way to advance significantly beyond dynamic verification is to increase the adoption of static verification. Leading-edge design teams have been using static verification successfully for a long time. It has been used strategically by designers to improve design quality and to complement dynamic verification on coverage closure. Static verification techniques help accelerate the discovery and diagnosis of design flaws during functional verification, reducing the time required to verify a design and simplifying the overall development cycle for complex SoC designs. In this article, we will summarize a variety of static verification techniques; including RTL lint, static RTL checks (which include low power structure verification and clock domain crossing verification), sequential formal checks, automatic formal applications, and assertion-based formal property verification.

As static analysis sees the RTL code and the synthesized representation of the design, it can naturally identify simulation and synthesis mismatches. By checking the synthesized representation of the design, it can highlight code segments that will cause synthesis to generate incorrect or inefficient logic. By analyzing the formal representation, it can identify potentially unreachable code and redundant functionality. All of this is important information, enabling designers to find better ways to rewrite the RTL code early in the design phase of the project.

Static analysis of gate-level netlists is also important. Synthesis and logic equivalence checking (LEC) have helped us move the majority of the design effect to the RTL. However, to design for test and design for power, many transformations are still performed on the gate-level netlist. Hence, it is still desirable to verify some specific functionality at the gate-level to ensure the transformations have been performed correctly and the functionality of the design has not been changed.

After a design has been fabricated, it is extremely difficult to debug any silicon related problems. Hence, anything that can be done up-front to prevent them will be extremely useful. Incrementally, based on the painful lessons learned debugging silicon failures, classes of static checks have been developed to look for these failure signatures at the RTL or gate-level. Two examples of this type of silicon failure are clock-domain crossing (CDC) metastability issue, and power domain isolation issue. Both of them cannot be verified efficiently with dynamic simulation. Instead, static verification can be performed to ensure design structures, such as synchronizers, are in place to safeguard against any metastability issues.

II. STATIC VERIFICATION TECHNOLOGIES

A. RTL Lint

RTL lint is a design and coding guideline checker. It checks HDL code for synthesizability, simulatability, testability, reusability, and RTL/gate signoff. The rules capture years of experience. Besides helping to enforce some known good naming schemes, they are designed to explore design and coding deficiencies that impact simulation, synthesis, test, and performance. RTL lint enables design reuse with prepackaged guidelines [1], such as the Reuse Methodology Manual (RMM), and STARC. Some of the common RTL lint rules include:

- Unsynthesizable constructs
- Unintentional latches
- Unused declarations
- Multiply driven and undriven signals
- Race conditions
- Incorrect usage of blocking and non-blocking assignments
- Incomplete assignments in sub-routines
- Case statement style issues
- Set and reset conflicts
- Out of range indexing

B. Static Checks

Static checking tools perform a pseudo-synthesis of the design into structural elements, such as registers, latches, combinational logic, finite-state-machines, and RAM. Most static checks are performed on this structural netlist. Hence, static checks can recognize any deficient or incorrect coding style for synthesis and, at the same time, identify any simulation versus synthesis mismatch issues. As the checks are no longer constrained by the module boundaries, they can explore connectivity issues, fan-in, and fan-out (driver-reader) relationships
within a design. However, it is important to realize that if part of the design is not synthesizable, it will be treated as a black box and static checks will not be performed. Some of the common static checks include:

- Combinational loops
- Full and parallel case issues
- Clock gating and usage issues
- Bus conflicts and floating bus
- Dead code or unreachable blocks
- Unused input and undriven output ports
- Unresettable registers
- Dead-end states
- Self-looping states
- Unreachable states
- Arithmetic overflow

Two of the static checks getting a lot of attention for today’s designs are clock domain crossing checks [2] and low power structure checks [3].

C. Formal Checks

Formal checks are performed on the formal netlist representation of a design. The formal representation incorporates the environmental constraints, the operational conditions, the design configurations, and the initialization sequence of the design. It represents how the design will operate clock cycle by clock cycle. Sequential design elements are represented in abstracted flow graphs; FSMs in state-transition graphs. Although formal checks have a lot of similarity with static checks, it does not focus only on structural connectivity, but also on functionality. The formal netlist allows the tool to calculate the possible values in the storage elements and nets of a design. As a result, it can foresee whether some scenarios are possible or not. For instance, if there are combinational loops in a design, formal checks can identify structural loops which are functionally impossible. This is useful as designers can focus on fixing the true combinational loops that actually can occur.

1) Functional coverage closure

Most designs have dead code, unreachable blocks, and redundant logic. This is especially true for IP or reused blocks. They often have extra functionality that is not needed for the current design. If passive coverage metrics, such as line coverage, FSM coverage, or expression coverage, are part of the closure criteria, unused functionality will have negative impact on the coverage number. Formal checks can be used to identify these unreachable blocks and redundant logic upfront, hence excluding them from the coverage measurement.

Especially with dead code and FSM analysis, formal checks are more comprehensive. They can identify signals which will stick at particular values. As a result, they will lead to functionally redundant logic. For instance, a user may specify that two inputs of a design are mutually exclusive. Then, by analyzing the functionality of the design, formal checking can identify which parts of their fan-out are functionality unreachable. Some other related formal checks are:

- Functional combinational loops
- Functional dead code
- Unreachable functionality
- Live and dead lock states
- Unreachable states
- Stuck at constant

2) X-state verification

Other uses of formal checks are related to x-assignments, x-propagation, and x-termination [5]. The goal is to eliminate pessimistic x-propagation as seen in simulation and to make sure an unknown or x-state is not generated or consumed unintentionally in the design. When an unknown state or an uninitialized state is sampled, the resultant value is unpredictable. Hence, it is also important to ensure that registers are initialized before they are used. Connecting a global reset to all the registers is ideal. However, due to routing congestion, it may not be always possible. The common formal checks related to X-state verification are:

- Reachable x-assignment
- Unguarded x-termination
- Uninitialized registers
- Use of uninitialized values

D. Automated Formal Applications

For some specific applications, formal verification can be applied to verify the structural correctness of a design. Similar to traditional formal property verification, assertions are used to capture the properties of the design. However, instead of being written manually, the assertions are generated automatically from their requirement specification. The verification process is completely streamlined.
1) **LEC constraint verification**

Logic equivalence checking (LEC) has been used extensively to perform RTL-to-gate level verification. It ensures that the gate-level netlist has implemented the RTL functionality correctly. To make sure the result is trustworthy, the assumptions used during the process need to be verified. FPV can help identify serious problems, such as conflicting assumptions and the violation of internal assumptions in library cells. Verifying functionality like these at the gate-level with simulation is extremely slow and it is impossible to cover all the possible scenarios. However, FPV will work well once the properties are derived from the gate-level netlist automatically.

2) **Timing constraint verification**

After the synthesis process, static timing analysis (STA) has been used extensively to identify the critical paths and to ensure the gate-level netlist has met timing requirements. STA works well on single cycle paths; however, not all of them are real, functional paths. Some of them are false paths (FP). In addition, STA can easily be misled by multi-cycle paths (MCP), paths between asynchronous clock domains, and paths between ratio synchronous clock domains. Hence it is important to identify these exceptions as constraints for STA and to ensure these constraints are specified correctly. Assertions can be generated to capture these FPs, MCPs, and CDC paths automatically. Then, FPV can be used to verify the correctness of these constraints.

3) **Circuit property verification**

As it is extremely difficult to debug any silicon related problem, anything we can do up-front to prevent them from happening will be very beneficial. One goal is to make sure the design meets the “cycle-repeatable operation” (CRO) criterion [4]. This means that the design can run a given test multiple times deterministically, with each run passing through the same internal states in the same order each time. In microprocessor designs, one of the requirements for CRO is to ensure one path is always selected in a pass transistor-based multiplexer. This is an easy property to capture in an assertion. Once it is done, FPV can be used to verify all of them in the design.

### E. Formal Property Verification (FPV)

For design teams to adopt FPV successfully on a design, we recommend they deploy FPV first on verification hot spots [6]. Verification hot spots are areas of concern within the design that are difficult to verify using traditional simulation-based methodologies. By focusing FPV on the hot spots, a design team can adopt FPV incrementally, side-by-side with their simulation-based methodology. We recommend each design team to focus on the verification hot spots in four design areas:

- Resource Control Logic
- Design Interfaces
- Finite State Machines
- Data Integrity

In addition, they can conduct an internal review to explore the unique hot spots in their designs. For formal property verification, design interfaces are popular verification hot spots among our customers. Interface verification using this approach has been employed successfully by a lot of our customers.

For verification purposes, we classify finite state machines (FSM) into two categories: interface FSMs and computational FSMs. Bus controllers and handshaking FSMs are examples of interface FSMs. Examples of computational FSMs are flow charts, data or control flow graphs. By capturing the properties of FSMs using assertions, we can identify them easily, exercise them completely and collect cross-product coverage information during the simulation process. When several FSMs are interacting with each other, it is important to ensure that they do not get “trapped” in a corner-case behavior. Formal verification can be applied to check this situation.

We also analyzed how the design teams added other assertions. Most of the teams spent significant effort on the resource control logic and data integrity hot spots. Assertions for resource control logic, such as arbiters, were easier to capture. Checkers from the assertion library were used extensively. Since most simulation environments did not stress test the resource control logic sufficiently, FPV did well and found a number of corner-case bugs. On the other hand, the assertions required to capture data integrity properties were complex, especially when data was reformatted, repackaged, or dropped intentionally. However, once these complex assertions were in place, they could be verified with various methodologies. They also represented an essential part of the verification plans; so the effort was well spent. Importantly, some of the bugs found were both critical and obscure. None would have been found with simulation alone.

### III. CONCLUSIONS

In this article, we discussed five prominent static verification methodologies that can be used to supplement dynamic verification. They allow verification to start early in the design cycle when designers are still working on the RTL code. Formal checks are unique in delivering both ease-of-use and accuracy. By automating the assertion creation process, we have helped deploy formal
property verification to target specific verification applications. Finally, 
verification hot spots are tried and true sweet spots for FPV. With 
formal technology, these static verification methodologies can examine 
all the possible scenarios in the design and find bugs missed by 
traditional simulation techniques.

As we have experienced with our customers, strategic deployment 
of static verification methodologies has proven to be effective. Static 
verification helps improve verification efficiency and boost the overall 
quality of the design.

REFERENCES


Freescale, DesignCon 2008.

[5] Mike Turpin, “The Dangers of Living with an X,” ARM, 

Verification methodology has undergone dramatic changes over the past decade. The realization that larger and more-complex designs required more and more verification effort, coupled with shrinking schedules, spawned new languages specifically tailored for verification and tools intended to make the verification process more predictable and efficient. As best practices were identified, languages and methodologies were standardized, and a supporting ecosystem of tools developed leading to the emergence of the coverage-driven verification flow.

In a coverage-driven verification flow, a stimulus model is described using constraints on randomly-generated variables. Describing stimulus in this way results in a stimulus description that is more concise, easier to review, and more productive to create than similar stimulus described with directed tests. An additional benefit of describing the valid stimulus space and using a tool to generate specific stimulus is that “surprising” stimulus is generated – stimulus that is valid according to the specification, but not necessarily anticipated by the verification engineer. This unexpected stimulus is critical to catching bugs in corner-case behavior earlier in the verification cycle, and is an added benefit of using automation.

The complete stimulus space for most designs today is so huge that there is no reasonable way to exhaustively exercise the stimulus space within a typical verification schedule. Consequently, a coverage model is constructed that identifies the key stimulus values, combinations of values, and sequences are required to exercise the design’s key functionality. Verification progress is measured by achieving the coverage described by the coverage model. This expanded set of verification metrics, and the verification management tools provided by leading verification tool flows, give management much-improved visibility into the verification process.

There are, of course, still challenges in the coverage-driven verification flow. One of the leading challenges is achieving coverage closure. In general terms, coverage closure means achieving the verification required by the verification plan. In a coverage-driven flow, this required verification is documented with the coverage model, which identifies the stimulus values, sequences, and combinations that must be applied to the design, as well as the design outputs and internal states that must occur. Achieving coverage closure means provoking the occurrence of each of the items specified in the coverage model.

When a random constraint solver is used to produce stimulus, it is easy to quickly produce large quantities of stimulus. However, because the stimulus is randomly produced, there is redundancy in the generated stimulus. To understand one reason why coverage closure is difficult to achieve with constrained random stimulus, consider the “Coupon Collector’s Problem” from probability theory. The subject of this problem is a game in which the object is to collect a full set of coupons from a limitless uniformly-distributed random collection. Early in the game, it is easy to fill empty slots in the coupon collection, since the probability is high that each new coupon selection is different from the previously-selected coupons. However, as the coupon collection approaches completeness, each new selection has a high probability of being a duplicate of a previously-selected coupon. After a bit of mathematical derivation, the expected number of selections needed to complete a set of \( n \) coupons is shown to be \( (n^2) \). Given a collection that contains 250 elements, we would expect to need to make around 1380 random selections to fill the set. With a typical-size coverage model, uniformly-distributed random stimulus results in a 10-20% stimulus efficiency rate. Put another way, coverage closure could be achieved 5-10 times faster if non-redundant stimulus were used.

Evolving the Coverage-Driven Verification Flow

by Matthew Ballance, Technical Marketing, SLE Division, Mentor Graphics

Figure 1 - Random Stimulus Coverage Closure Progress

While redundant stimulus poses a distinct and quantifiable challenge to coverage closure, the mismatch of stimulus space and coverage model poses an even greater challenge, one that is difficult to quantify. The declarative stimulus description typically describes the entire valid stimulus space. In the case of the description for a network packet,
for example, the stimulus description encompasses the total space of valid packet sizes – perhaps 1 byte through 16 kilobytes. From a coverage perspective, however, a smaller space is of interest. A verification engineer typically identifies specific key values, out of the larger valid space, to monitor for coverage purposes. In the case of our network packet, very small packets, medium-size packets, and very large packets might be identified as being of interest from a coverage perspective. Unfortunately, the stimulus generation process has no knowledge of this more-selective coverage goal. Stimulus continues to be evenly generated across the entire domain of 16,384 valid packet sizes, even though only a much-smaller subset is meaningful from a coverage perspective. Consequently, the probability of hitting the desired coverage is dramatically reduced. The biggest problem with the mismatch between the stimulus space and the coverage model isn't actually the increased time to achieve coverage. It is the uncertainty and unpredictability of achieving coverage. It's difficult enough to squeeze the required verification for one of today's complex designs into the verification schedule. Not even knowing how much time to budget for a given amount of functional coverage makes schedules very inexact.

Several approaches are currently used to reduce the negative effects of redundancy and stimulus/coverage model mismatch on coverage closure. One of the easiest and most automatic is to vary the seed used by the random constraint solver. The seed used by a pseudo-random generator influences the sequence of generated data. In theory, running many short simulations, each with a different seed, should result in accelerated coverage closure because each simulation targets a different portion of the stimulus space. In practice, however, the relationship between a seed and the stimulus produced is opaque and difficult to understand. To revisit the Coupon Collector’s Problem, this approach is similar to shopping at multiple stores when collecting coupons. Yes, the random selection available at each store is slightly different. However, the end result is only a slightly-faster rate of collecting a full set of coupons.

A commonly-used approach that results in better coverage-closure acceleration is to manually analyze the holes in coverage and appropriately bias the random generation to target these holes. This approach can dramatically improve coverage closure when a mismatch between the stimulus space and the coverage space is the primary barrier to coverage closure. In the case of the packet generation example above, we might bias the random constraint solver towards the packet lengths that our coverage model targets. However, this approach requires significant human effort to analyze the coverage and stimulus models and design an appropriate biased-constraint test to target a specific set of missing coverage. Time spent targeting stimulus is time that could be more productively spent creating more tests or investigating test failures.

Any repetitive labor-intensive process is a candidate for automation, and stimulus coverage closure is no exception. A key realization is that the manual effort in stimulus-coverage closure is often expended to align the stimulus model with the coverage model. Using a tool to guide stimulus production according to coverage goals – so-called closed-loop verification or coverage-driven stimulus – represents a powerful evolutionary step in the coverage-driven flow. By allowing stimulus generation to be tailored to the coverage model, coverage closure can be achieved much more quickly than with unguided random stimulus. In addition, a tool automatically performs the analysis of coverage model and stimulus model to arrive at an optimal set of stimulus, so human effort needn't be spent on this tedious task.

Mentor Graphics inFact is a coverage-driven stimulus generator with some very unique capabilities that result from the tool’s knowledge of both the stimulus and coverage models. inFact accepts a compact declarative description of the stimulus space. During simulation, inFact algorithms efficiently target user-specified portions of the coverage model. One of the key capabilities of inFact is the ability of the algorithms to control redundancy in the generated stimulus according to the coverage model. Stimulus that is significant according to the coverage model is generated without redundancy, while stimulus that is not significant from a coverage perspective continues to be generated randomly. This, alone, makes stimulus generated by inFact’s algorithms an order of magnitude more efficient at achieving coverage closure than random stimulus that is generated without knowledge of the coverage model.

A coverage model encompasses multiple independent coverage goals. inFact's algorithms take a macro view of these multiple independent goals and generates stimulus that efficiently targets these goals simultaneously. In this way, inFact provides superior results when compared to a manual approach to biasing random generation, which typically can only target a limited number of coverage goals during any one simulation.
Automatic and efficient distributed verification is another key benefit that inFact delivers because of its knowledge of both the stimulus and coverage models. Because inFact is aware of the stimulus model and the portions of the coverage model targeted during a particular verification session, it is able to dynamically divide up the stimulus required to achieve the specified functional coverage goals and distribute that stimulus across multiple simulation sessions running on a simulation farm. When running in this manner, inFact is able to eliminate redundant stimulus not only in one simulation but across simulations running on many machines in parallel. inFact offers an order-of-magnitude boost in simulation productivity in a single simulation. This benefit scales linearly as more machines are added to the distributed verification session. The end result is a huge boost in throughput by efficiently leveraging distributed computing.

Coverage-driven verification has revolutionized hardware verification, raising test-creation productivity, providing management with better visibility into the verification process via verification metrics, and increasing design quality. Evolving the coverage-driven verification flow to include coverage-driven stimulus dramatically increases the productivity of verification simulation, permitting coverage closure to be achieved an order of magnitude faster than with random stimulus generation and without using human effort to analyze coverage results and direct the stimulus model. This, in turn, leads to more predictable verification schedules, more meaningful verification being achieved and, ultimately, higher-quality results.
OVERVIEW

The Ethernet Multi-view Component (MVC) is an OVM-based Verification Intellectual Property (VIP) that eases the verification process for interfaces defined in IEEE 802.3-2008, part of the family standards defining the Physical Layer (PHY) and Data Link Layer’s media access control (MAC) sublayer of wired Ethernet.

Ethernet MVC provides a SystemVerilog interface for hooking a design under test (DUT) at various interfaces:

- CGMII, XLGMII, XGMII, GMII and MII
- 100GBASE-R, 40GBASE-R, 10GBASE-R, 10GBASE-X, 10GBASE-W and 1000BASE-X (both serial and parallel mode)
- Auto-negotiation for backplane Ethernet
- Standalone PHY
- Management interface
- Non IEEE standards RTBI, RGMII and RMII

Consider wired Ethernet from the MAC through Physical Medium Dependent (PMD) layers: The MAC layer is responsible for framing the packet coming from upper layer; it primarily adds the checksum field to the packet received from upper layer. The Reconciliation layer puts the frame byte-wise on xMII interface. The resultant frame is sent to the Physical Coding Sublayer (PCS). The PCS layer then performs complex functions like 8b/10b or 64b/66b encoding/decoding and/or scrambling/descrambling; it also deskews the lanes and achieves synchronization on lanes at the receiving end. The Physical Medium Attachment (PMA) layer does serialization and de-serialization.

The PMD sub layer is the interface to the transmission medium. Of these, the PCS layer is associated with the most verification complexity.

Ethernet MVC reduces this complexity in at least two ways: first, by providing extensive configurable and error-injection features at the PHY level; and second, by providing transaction viewing at various abstraction levels, which considerably reduces verification time.

This article concentrates mainly on a strategic approach for PHY verification. For this, we will first describe using the MVC architecture to plug-in the DUT. This section describes various building blocks used in MVC. Then we will discuss using the MVC environment to plug in the PHY DUT and address the role of the scoreboard in PHY verification. Finally, we will present how to exploit the configurable features of Ethernet MVC to verify critical functionalities of the PCS layer.

MVC ARCHITECTURE

With a very simple architecture, Ethernet MVC provides a SystemVerilog interface that is used to hook-in the DUT. The test bench architecture requires very few connections of ports and exports of the transaction level modeling (TLM) components to get the test up and running. In some cases MVC can take care of the connections of ports and exports itself, requiring only the instantiation of the components. The basic architecture of an Ethernet MVC is shown below:
When the DUT has a MAC interface, it implements the MAC layer and Reconciliation layer. If the DUT has a PHY interface, then it must implement MAC and Reconciliation layers and also the PHY sublayers.

Depending on the type of DUT to be verified, the SystemVerilog interface can be configured for any MAC interface (i.e., [C][XL][X][G]MII) or any PHY interface like BASE-X or BASE-R. The agents at either end of the bus play an active part in the protocol. Each of these agents can be configured to be in active or passive mode. An agent in active mode automatically instantiates the sequencer, driver and monitor components, and is responsible for running sequences that may be targeted to a DUT. In this architecture, a sequence is shown to be running on the MAC agent side. This sequence is meant to send or receive frames going to or coming from the DUT. On the other hand, an agent in passive mode does not instantiate sequencer or a driver; it simply monitors the transactions. In this architecture, PHY agent is assumed to be in passive mode. The monitor inside this agent observes transactions on the bus and publishes those transactions for use inside or outside the agent. When the MAC and PHY agents are in passive mode, both will be in monitor mode.

The inbuilt coverage collector is part of MVC’s agents and is primarily used for frame and symbol-level functional coverage of the MAC and PHY interfaces.

**MVC ENVIRONMENT TO PLUG-IN THE PHY DUT**

A PHY DUT has two physical interfaces: a MAC side interface and a PHY interface. Two instances of MVC are required to hook-in this DUT. One interface should be configured for the MAC interface, such as XGMI, and the other interface should be configured for the PHY interface based on the type of DUT (BASE-X or BASE-R). Figure 2 shows the environment for plugging in 10GBASE-X PHY.

This environment also demonstrates the role played by Scoreboard in PHY verification.

Here Scoreboard compares the frames coming in and out of PHY in both the transmit and receive directions. If there happens to be an error, miscompare would return something like:

"OVM_ERROR @ 1508000: env.ith_scoreboard [eth_sb_id] Frame retrieved from XAUI TX interface does not match with one retrieved from MAC TX interface"

The scoreboard has two analysis exports for each direction to analyze the inflow and outflow of frames from PHY. Mismatch between the transactions captured at either end of PHY produces an error message. Such a mismatch can occur due to an erroneous behavior of PHY in the processing frame, and so the scoreboard essentially is a starting point for error detection and subsequent debugging.

**PROBLEM AND STRATEGIC APPROACH FOR PHY VERIFICATION**

The complex functions of the PCS layer are split into three categories:

**Encoding/Decoding**

Encoding and decoding of symbols is among the major PHY tasks. This can be 8b/10b (in the case of BASE-X) and 64b/66b (in the case of BASE-R). MVC helps ensure that PHY encoding/decoding functionality works perfectly.

MVC automatically encodes symbols by implementing transmit-side state machines; by using directed stimulus, it’s possible to generate an exhaustive list of encoding scenarios. MVC also allows fine control over insertion of transmit-side errors that can be used to target specific DUT functionalities. For instance, in BASE-R, individual fields like sync header and block type field of a particular 66b block can be corrupted. Similarly, in BASE-X, errors like disruption of disparity value can be inserted. The net result is that the verification engineer should easily be able to check the symbol decoded by DUT. In this scheme, the coverage collector at the PHY interface plays a vital role. The coverage collector shows 100 percent coverage only after hitting all possible bins as per the verification plan. The verification plan includes an extensive list of coverage points such as reception of all data and special code-groups (with positive and negative disparity) etc.
Some code snippets for sending valid and invalid frames:

To send a randomized data frame on the bus with no error
(irrespective of BASE-X or BASE-R PHY):

```cpp
ethernet_data_frame data_frame =
    ethernet_data_frame::type_id::create("data_frame");
assert(data_frame.randomize());
start_item(data_frame);
finish_item(data_frame);
```

To send an erroneous data frame:

```cpp
eth_cfg.m_dev0_cfg.m_dyn_eth_cfg.m_basex_set_disparity_error[0] =
  ETH_BASEX_ERR_IN_COL_S;
eth_cfg.m_dev0_cfg.m_dyn_eth_cfg.m_basex_disparity_error_on_lane[0]=3;
ethernet_data_frame data_frame =
    ethernet_data_frame::type_id::create("data_frame");
assert(data_frame.randomize());
start_item(data_frame);
finish_item(data_frame);
```

The configuration settings before running the frame specifies that
disparity error should be inserted in lane 3 when a ||S|| ordered set is
being transmitted.

**Scrambling/Descrambling**

For the serial interface, scrambling is intended to provide the DC
balance and sufficient transition density of bit stream transmitted on
the medium. Thus scrambling is the vital feature to be verified. MVC
does scrambling and descrambling automatically. On the MAC side of
the DUT, MVC captures the frame from blocks descrambled by the DUT. If the DUT does not descramble the frame properly, MVC will not
capture the complete frame, an outcome that will be apparent in the
transaction view of MVC. Similarly, on the PHY side of the DUT, MVC
will descramble the blocks that have been scrambled by the DUT. If
there is an error in the scrambling logic of the DUT, MVC will not be
able to synchronize.

MVC also provides an option to bypass the scrambler at the
transmitting end and similarly bypass the descrambler at the receiving
end. This feature can be exploited to do stepwise verification.

Since it is impossible to associate any meaning with the scrambled
block, the verification may begin with turning off the scrambling in the
DUT. This would require turning off the descrambler on the receiver
side of MVC and turning off scrambling at the transmitting end of MVC.
All of this provides flexibility for stepwise debugging of issues other
than scrambling.

Some code snippets for configuring MVC for some scrambling
features:

To bypass scrambling:

```cpp
config_class.m_static_ethernet_config.m_baser_bypass_scrambler = 1'b1;
set_config_object("\*.top_cfg_id",config_class);
```

(Note this is static configuration and needs to be done prior to
running the stimulus.)

**Synchronization and Deskew logic**

The XAUI interface has four serial lanes, each carrying serialized
10-bit symbols. Apart from data, these symbols carry special code-
groups inserted by the PCS layer. These can be /K/, /A/ and /R/ for
Idle code-group pattern. For this text, we are concerned with the role
of Align symbol (/A/) and sync symbol (/K/). The sync symbol is used
to achieve code-group synchronization. This is done by detecting
code-group boundaries in the incoming bit stream of each lane. A
simple example helps illustrate the point:

The transmitter transmits the following bit sequence on four lanes
(K0 to K9 are 10 bits of /K/ whereas R0 to R9 are 10 bits of /R/):

<table>
<thead>
<tr>
<th>LANE0</th>
<th>LANE1</th>
<th>LANE2</th>
<th>LANE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
</tr>
</tbody>
</table>

But when this reaches the receiver, lane 3 gets skewed by five bits

<table>
<thead>
<tr>
<th>LANE0</th>
<th>LANE1</th>
<th>LANE2</th>
<th>LANE3</th>
</tr>
</thead>
<tbody>
<tr>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
<td>K0 K1 K2 K3 K4 K5 K6 K7 K8 K9 R0 R1 R2 R3 R4 R5 R6 R7 R8 R9</td>
</tr>
</tbody>
</table>

The PCS receiver must then be able to achieve synchronization by
detecting COMMA symbols and achieving code-group alignment. To
test this logic of the DUT, MVC’s transmitter can be configured to insert
a desirable skew at the bit level on any lane. After the DUT’s PCS receiver is verified for the PCS synchronization state machine, the next step is to verify deskew logic at code-group level. The transmitter inserts align column (||A||) at regular intervals on each column. Due to the inherent nature of medium and associated devices, the receiver may not see /A/ symbols at the same time on every lane.

An example illustrates the problem:

The transmitter transmits the following symbol sequence on four lanes (K is sync, R is Skip, S is Start, Dp is preamble, Ds is SFD). Note that /A/ is transmitted on each lane simultaneously:

LANE0: K R A K R K R K R S Dp D ...
LANE1: K R A K K R K R K R Dp Dp D ...
LANE2: K R A K K R K R K R Dp Dp D ...
LANE3: K R A K K R K R K R Dp Ds D ...

But when this reaches to receiver, the lane 2 gets skewed by three symbols resulting in absence of the /A/ code-group on one or more lanes in a column:

LANE0: K R A K K R K R S Dp D ...
LANE1: K R A K K R K R K R Dp Dp D ...
LANE2: K R A K K R K R K R Dp Dp D ...

In such situations, the PCS receiver cannot put the symbols on the XGMII interface on an as-is basis. It must deskew the lanes so that it puts the ||A|| ordered set at any clock edge of the XGMII receive interface; only then can the MAC layer successfully receive frames. To test this logic, the user can configure the MVC transmitter to insert a skew at the symbol level on any lane. With this, the deskew logic of DUT’s PCS layer can be easily verified. On the other hand, MVC’s receiver also does the deskew operation on the receive side.

Next, consider how to write intelligent test cases to check state transitions of the DUT receiver. The PCS deskew state machine (Figure 48-8 of IEEE Std 802.3-2008) transitions from LOSS_OF_ALIGNMENT state to ALIGN_DETECT_1 state on receiving ||A||. The machine makes the transition to LOSS_OF_ALIGNMENT state on detection of deskew error (that is /A/ recognized in fewer than all lanes). These transitions can be checked by writing the following directed test case:

```
ethernet_data_frame data_frame =
  ethernet_data_frame::type_id::create("data_frame");
ethernet_idle idle = ethernet_idle::type_id::create("idle");
// Initiating frame of maximum size would ensure that idle column
// immediately after ||T|| column would be ||A||
assert(data_frame.randomize() with { data_frame.len_type_field == 1500; });
start_item(data_frame);
finish_item(data_frame);
// This idle must be ||A|| and should cause state transition from
// LOSS_OF_ALIGNMENT to ALIGN_DETECT_1
start_item(idle);
finish_item(idle);
// Configure the MVC transmitter to insert error in next ||A||
// by setting invalid symbol on lane 3
eth_cfg.m_dev0_cfg.m_dyn_eth_cfg.m_basex_set_invalid_symbol[0] = ETH_BASEX_ERR_IN_COL_A;
eth_cfg.m_dev0_cfg.m_dyn_eth_cfg.m_basex_invalid_symbol_on_lane[0] = 3;
// Run some idle columns
repeat(32)
begin
  start_item(idle);
  finish_item(idle);
end
```
CONCLUSION

We discussed a simple architecture that is used in Ethernet MVC. This architecture demonstrated that the MVC environment can easily be created with minimal user intervention. All the instantiations of driver, sequencer, monitor and coverage collector, and so on are automatically taken care of by MVC, and just require simple configuration settings. Next, we discussed the environment where two instances of MVC were made to plug-in the PHY DUT with two physical interfaces. The role of Scoreboard in catching the problems in the PHY DUT was discussed. We then looked at various functions in the PCS layer for which verification must be targeted. These were encoding/decoding, scrambling/descrambling, synchronization and deskew logic. We demonstrated a strategic approach to be taken in stepwise verification. The role of error insertion in testing specific functions in PCS layer was discussed.

REFERENCES

1. IEEE Std 802.3-2008 specification
3. MVC Library Documentation
ABSTRACT

This article discusses, from a Design Engineer's perspective, the ease and benefits of migrating to a Verification environment using Questa and OVM Methodologies from a traditional native testbench environment using HDL's.

Considering the current economic constraint, today's VLSI engineers need to wear multiple hats, right from architecture design, Implementation to Verification. Migrating from a customized Hardware design environment to a standard Hardware Verification environment may seem trivial, but considering the advanced features/ additional capabilities (as compared to HDL) within an Object Oriented Context provided by HVL such as SystemVerilog with the latest Verification Methodology/tools (Questa/OVM) certainly proved challenging but also beneficial.

The initial challenge of migrating from Verilog to SystemVerilog was trivial as it is an extension to the former except within an object-oriented context. The most valuable feature of OVM found other than the re-usability factor is the ease of use with which the complete verification environment was developed considering the large set of components (base classes) readily available to generate basic building blocks such as the stimulus generator, sequencer, driver and scoreboard. It was just a matter of thinking in terms of how and what needs to be verified rather than writing verification components from scratch.

As far as the design engineer is concerned, we are more interested in directed test benches and test cases that we think of in terms of verifying the design (module or chip level). However, as the complexity of the chip is increasing, there is much more than one can think of which can only be found using a constrained random verification environment like we developed with SV/OVM.

Another important factor from the design engineer’s view is Code coverage, which used to be enough to sign-off a design. However as the complexity has increased, functional coverage analysis has emerged as the true winner in determining verification closure.

OVM/Questa answers all these needs and much more. OVM provides a ready-made and easy-to-use solution to set up a complex, reliable and re-usable verification environment. Also Questa provides advanced features such as integrated verification management tools, transaction viewing and debugging which helps achieve verification plan targets.

Considering all the achievement of methodologies and tools, the actual benefit comes from the ability of the design engineer to use these verification environments in the early stages of RTL development. Often designers do not rely on the HVL based verification environment when they are in the early stage of development since many times they are not directed enough to satisfy the design engineer’s very specific needs. The only intention of a design engineer at the module level stage is simple correctness of his/her design. However, as per our experiment, such an environment can provide added benefits in finding the bugs early since automated randomization would enhance directed testing early on, thus narrowing the time-to-market window. Efficient planning (functions/signals that need to be verified) and advanced tool features should be taken into account while developing the verification environment to enable chip level verification.

In this article we will be discussing a simple example; The Creation of an OVM based Verification environment for a GEMAC IP. We will explain the verification strategy using OVM and SystemVerilog and describe the challenges that were met in the generation of a standard verification environment focusing on the benefits achieved as well as the performance enhancements along with the ease of use.

CHALLENGES IN VERIFICATION

Typically a design engineer is more familiar with design concepts and design implementation challenges and is accustomed to using HDLs. They may not be familiar with verification concepts and complexity in verification. Most of our thoughts are very limited relative to the design in terms of functional/simulation faults. We are aware of the corner cases (or rather think that we are!) and flaws in the design, but of course that is usually an over-statement. A verification engineer’s primary intention is to break the code, but he may not catch these known issues until they are specified or there could be a significant delay to create an environment and corner them. Of course there will also be a number of unpredictable bugs.

When the design is in progress, a design engineer may need to check some particular scenarios at the block level. At the same time, verification engineers do not want to spend their time writing only
directed test cases; also they wouldn’t have to come up with entire verification environment. It may cause a significant delay later and considering time-to-market constraint it would sound beneficial to have the environment ready first-hand and add on to it incrementally.

So how would a design engineer scale-up to the verification engineer’s level?

According to our experience in the GEMAC IP Project, there were huge challenges in the areas discussed below, keeping the big question in mind, “is my design fault free?”

• Have an integrated environment that both design and verification engineers can use for system level or block level testing. The challenge here for us was to keep up with the robust designs and changes/additions to features that keep happening within the projects. The test environment should be reusable and independent to accommodate any such modifications and additions required.
• Have a comprehensive test plan and an automated regression process that submits thousands of tests to all possible scenarios and automated reporting of each test and bugs for failures.
• Reports should be presented graphically that show statistics of tests (passed, failed, crashed etc) and for the bugs over time.
• Developing tests that verify the correctness of the design, like self checking randomization.
• Layering of sequences, so as to reduce the lines of code and improve reusability.

You just cannot think of easily accommodating these in your traditional verification environment

SOLUTIONS USING OVM

OVM provides a full-fledged solution for verification. It has a rich set of base classes and other infrastructure which makes for a more efficient verification environment. A design engineer does not need to be an expert in object oriented concepts to come up with a basic environment.

Abstract classes are available for all verification components. Once you understand the structure of OVM based VIP, it is very easy to create a basic environment and can be enhanced it by adding features without disturbing existing ones. The biggest advantage is you do not need to start from scratch as in case of other environments. One logical element will be sufficient to put the components in order.

Our GEMAC VIP has three agents in one environment. We started with gemac_host_agent then created the others. Although block level verification is different from top-level verification, using an OVM environment it was easy to develop a top-level environment from block level, by reusing many of the verification components. Most of the changes required were in development of the driver component module.

```
class gemac_host_agent extends ovm_agent;
    ...
    function void build();
        super.build();
        monitor = gemac_host_monitor::type_id::create("monitor", this);
        if(is_active == OVM_ACTIVE)
            begin
                $cast(driver, create_component("gemac_crc_driver", "driver"));
                driver.build();
                ....
            end
    endfunction
endclass
```

The method set_type_override() allows for reuse of components to check the top-level functionality by overriding the driver component without altering other components.

```
class gemac_test extends ovm_test;
    gemac_env env;
    function void build();
        ovm_factory::set_type_override("gemac_crc_driver", "gemac_host_driver");
        ....
    endfunction
    ....
endclass
```

Verbosity level setting is yet another feature; the best thing is that the verbosity can be set from the script file also. All these features make OVM more user-friendly and make debugging quite easy.

A very prominent feature of OVM is sequence layering. It enables easy creation of any number of sequences using basic lower level sequences. You just need to try different combinations to make different higher layer sequences and scenarios. In the case of GEMAC VIP most of our efforts for generating different scenarios were significantly lowered by making a virtual sequence to serve as a “mastermind.”
Among the three agents, gemac_mgt_agent is only responsible for configuring the DUT. It was very difficult for us to verify the DUT at different CSR configurations using traditional verification methods. For each test, you have to reconfigure the DUT manually. OVM’s virtual sequencer with macro `ovm_do_on and `ovm_do_with provided a new solution to this problem of which we can use various combinations of sequences from different sequence libraries.

```plaintext
class virt_seq_1 extends ovm_sequence;
  `ovm_sequence_utils(virt_seq_1, gemac_virtual_sequencer)
  host_normal_pkt_seq_1 host_normal_1;
  host_vlan_pkt_seq_1 host_vlan_1;
  mgt_seq_2 mgt2;
  mgt_seq_8 mgt8;
  mgt_seq_33 mgt33;
  gmii_normal_pkt_seq gmii_normal_1;

  function new();
    host_normal_1 = new();
    host_vlan_1 = new();
    mgt2 = new();
    mgt8 = new();
    mgt33 = new();
    gmii_normal_1 = new();
  endfunction

  virtual task body();
    `ovm_do_on(mgt8, p_sequencer.mgt_seqr)
    `ovm_do_on(host_vlan_1, p_sequencer.host_seqr)
    `ovm_do_on(mgt2, p_sequencer.mgt_seqr)
    `ovm_do_on(host_normal, p_sequencer.host_seqr)
    `ovm_do_on(mgt33, p_sequencer.mgt_seqr)
    `ovm_do_on(gmii_normal_1, p_sequencer.gmii_seqr)
  endtask
endclass : virt_seq_1
```

Each `ovm_do_on macro will execute sequentially. It is also possible to execute the sequences in parallel. A simple fork/join with macros `ovm_do or `ovm_do_with will serve this purpose. There is another method start() from OVM that can also be used to execute the sequences in parallel. The diagram below shows how a virtual sequence takes control of driver sequencers and executes the sequences from different sequence libraries.

When we first thought of developing the GEMAC OVM VIP for our GEMAC IP, our first thought was; are we using a most complex weapon to kill a housefly? The library was quite comprehensive with a huge number of methods and base classes in OVM. It was also different from the traditional test environment using Verilog, heavily relying on object-oriented concepts. At the beginning it was rather unclear how to generate a particular test. However as we progressed with our work, we realized that this most sophisticated weapon could also be customized to achieve our goal. The difficulties that we faced during this development were soon surpassed by the advantages that we got using OVM. The OVM User Guide and Reference Manual were very helpful and also the examples helped us to understand the concepts more clearly.

**SOLUTIONS USING QUESTA**

The Verification Plan is the golden reference that a verification engineer is constantly watching and against which a design engineer must verify specific functionality. If any tool imports the verification plan as an input file, then of course the result of that will be the topic of interest for a design engineer.

Verification Management with Questa makes the distance to reach the verification goal shorter. The set of functionality exhibited reduces the burden on the design engineer by providing efficient graphical reports and statistical view of the database.

Our GEMAC VIP has hundreds of scenarios and a number of test cases. After merging 3-4 test results, we got to see the following results. It shows the Browser tab within the Verification Management window that displays summary information for merged results in a UCDB for GEMAC verification plan.
From this browser, it is easy to figure out the design faults and extra logic that a design engineer has created unintentionally. It enables the design engineer to have a closer look at his design.

The report as shown below gathers the information on coverage of the block level as well as the overall environment. Using this kind of analysis, a design engineer can determine the code coverage and further enhance the environment for maximum throughput.

Functional Coverage information is given in the diagram below; it makes both verification and design engineers more conscious about their targets achieved and flaws in design. The statistical view gives the better feeling and easy understanding for the engineers rather than the tiring view on reports using traditional HDL test environments.

**CONCLUSION**

Developing an OVM Verification environment can be challenging for a design engineer. However, the numerous faults that were detected and corrected using the environment showed that it was definitely worth the effort. Traditional HDL has its limitation in terms of how much and how efficiently a design can be verified. OVM and Questa together provide an efficient and reliable way to achieve verification goals.

OVM makes it easy to reuse and enhance existing code. Questa offers efficient debugging and reliable reports, which reduce the efforts on both design engineer as well as verification engineer. Undoubtedly, both our design and verification engineers found that Questa and OVM served their needs quite well. We strongly urge you to consider taking advantage of them yourself.
1. INTRODUCTION

Scoreboarding is a fairly straightforward concept used in functional verification environments.

Simply put, when an event of interest is anticipated by the verification environment, details of that event are posted to the scoreboard. Conversely, when an event of interest is actually observed, it is checked against the events already posted on the scoreboard. The mapping of the posted to the checked events is called the transfer function, which can range from the fairly straightforward to the fairly complex. For example, for every posted event, there may be multiple events that are checked and vice versa.

A verification project typically requires the following features from its scoreboard implementation:

• In-order and out-of-order checking
• Timeout checking
• Hooks for error handling
• Handling of expected dropped transactions
• Support for complex transfer functions
• Support for both procedural- and port-based posting and checking of events

Interestingly, the latest OVM 2.0.2 release does not provide a generic implementation of scoreboard that supports these features. While there is a base class called ovm_scoreboard, it is left up to the user to implement its functionality. Users end up creating home-grown versions of the scoreboard implementation, and we feel a generic scoreboard class implementation is in order.

To address this need, we have contributed the SystemVerilog FrameWorks™ Scoreboard (SVF Scoreboard) package, an open source implementation, to the OVM World website[1]. The contributed package is implemented in SystemVerilog and supports all the features listed above. To date, there has been more than 450 downloads by the OVM user community at large and it is currently being used by several of our own clients. In this article, we discuss some of the features of the SVF scoreboard functionality and show how the verification engineer can quickly integrate scoreboard into their verification environment. We encourage the reader to download and explore this contribution and welcome feedback [2, 3].

2. SVF SCOREBOARD USE MODEL

In this section, we present two typical use models of the SVF scoreboard: a simple and an advanced use scenario. These are generic scenarios and can be customized to any extent depending on the needs of the project.

Figure 1 illustrates a simple use scenario, where an instance of the SVF scoreboard can be pretty much dropped into an existing environment. The SVF scoreboard in Figure 1 is an instance of the pw_scoreboard class and is derived from the ovm_scoreboard class. The ovm_scoreboard is an empty built-in base component in OVM library. Transactions are posted to the scoreboard as instances of classes derived from the ovm_transaction class. Similarly, transactions are checked by passing instances of classes derived from ovm_transaction to the scoreboard. The actual comparison takes place by calling the ovm_compare method of the posted object with the instance of the checked object as an argument.

![Figure 1. A Simple SVF Scoreboard Use Model](image)

In the simple use model, the transactions are being sent from the driver, and expected to be transmitted as driven in a simple in-order fashion through the design under verification (DUV). A monitor sits on the stimulus side, and publishes observed stimulus to its analysis port. The analysis port is connected to the scoreboard and the expected values get posts whenever a transaction is observed. For its counterpart, another monitor sits on the response side, which publishes the observed transactions. When a DUV response is published by this monitor, it gets checked against the posted values in order.
Figure 2 shows an advanced use model of the SVF Scoreboard. Here, the mapping between the posted stimulus to the expected response is not so straightforward and requires some manipulation of both the stimulus and the response data to infer the actual data being checked. As an example, consider a packet driver that sends in a sequence of packet fragments. The predictor collects the fragments being sent and posts the complete packet data to the scoreboard once it has seen the end of the packet. On the other side, the response monitor sees the outgoing fragments and sends them to the checker. When the checker finally assembles the entire packet, it sends it to the scoreboard for comparison. However, the checker also gets the input from the error monitor and marks whether an error occurred so that the affected data can be compared appropriately with the posted value. This approach of separating the prediction and checking transfer functions in a separate component offers a powerful and generic methodology for scoreboard. The details of the predictor and checker are described later in Section 4 (Advanced Transfer Functions).

3. MULTI-STREAM POSTING AND CHECKING

The SVF scoreboard supports the notion of an arbitrary number of streams (Figure 3). In-order checking is accomplished by posting objects to the same stream. Out-of-order checking is accomplished by posting objects to different streams. Each stream is identified by a unique number, and objects posted to a given stream are expected to be checked in the same order as they are posted. There is no implied order between objects posted in different streams. Thus, objects can appear in any order with respect to each other if they are posted in separate streams.

4. ADVANCED TRANSFER FUNCTIONS

When data is transmitted from one interface to a different kind of interface, a complex transfer function may be required to represent the relationship between one data type to another data type. Although the transfer function will be very much design specific, it can still be done in a consistent and systematic manner. By providing hooks to allow design specific transfer functions, the scoreboard can be highly reusable.

Figure 4. Implementing predictor transfer function

Figure 5. Implementing checker transfer function
SVF scoreboard provides a `pw_predictor_checker` class to handle various transfer function(s) between data being transmitted and data being received. Figure 4 and Figure 5 respectively illustrate how the predictor and checker objects implement complex transfer functions. The predictor object may receive stimulus data from multiple sources using its exports. Depending on the application need, it then infers the appropriate data and posts it to any of the connected scoreboard instances as needed. On the other hand, the checker object, analogously, receives observed response data from multiple sources through its exports, and then, as the application dictates, forwards the inferred response data to the appropriate scoreboard for checking.

Figure 6. Example Showing Use of SVF Scoreboard.

5. EXAMPLE

Figure 6 depicts a typical example of connecting the SVF scoreboard to a testbench via the TLM interfaces. In this example, two `pw_predictor_checker` class instances are created. The instance connected to the monitor at the input side (left) of the DUT works as the predictor. The predictor is responsible for converting the input data type to the expected output data type. The instance connected to the monitor at the output side (right) of the DUT works as the checker. The checker is responsible for comparing the observed output data with the expected data on the scoreboard. The predictor connects to the `pw_scoreboard`'s `post_export` and the checker connects to the `check_export` of the scoreboard.

The following shows the example code snippets for setting up the scoreboard as shown above:

Step 1. Declaring a scoreboard and two `predictor_checkers`.

```plaintext
class xbus Demo_env extends ovm_env;

// Instantiate a scoreboard. Both input and output data
// are of type xbus_transfer
pw_scoreboard #(xbus_transfer, xbus_transfer) pw_sb;

// Instantiate a predictor. Number of input ports is '1',
// number of output ports is '1'. Input and output data
// types are both xbus_transfer
pw_predictor_checker #(1,1,xbus_transfer,xbus_transfer)
    pw_predictor;

// Instantiate a checker. Number of input ports is '1',
// number of output ports is '1'. Input and output data
// types are both xbus_transfer
pw_predictor_checker #(1,1,xbus_transfer,xbus_transfer)
    pw_checker;

endclass
```

Step 2. Connecting the components

```plaintext
class xbus Demo_tb extends ovm_env;

virtual function void build();
    super.build();

    pw_sb = pw_scoreboard#(xbus_transfer,
        xbus_transfer):type_id::create("pw_sb",this);
    pw_predictor=pw_predictor_checker#(1,1,xbus_transfer,
        xbus_transfer):type_id::create("pw_predictor",this);
    pw_checker=pw_predictor_checker#(1,1,xbus_transfer,
        xbus_transfer):type_id::create("pw_checker",this);

endfunction

endclass
```
Step 3. Connecting checker/predictor and scoreboard ports

```verilog
class xbus_demo_tb extends ovm_env;
    function void connect();
        ...
        xbus0.master[0].monitor.item_collected_port.connect(pw_predictor.inp_exports[0]);
        xbus0.slaves[0].monitor.pwItemAtCollected_port.connect(pw_checker.inp_exports[0]);
        pw_checker.sb_aports[0].connect(pw_sb.post_export);
        pw_predictor.sb_aports[0].connect(pw_sb.check_export);
    endfunction
    ...
```

Step 4. Defining the transfer function

```verilog```
// Transfer function that processes the arrived transaction
// Also specifies the port id at which it arrived
virtual task transfer(T_INP trans, int port_id);
    ovm_report_message("pw_predictor_checker", $psprintf("SB: transfer: %d
", port_id));
    // Do what you need to do with trans
    // Push it to the SB port (port_id % NUM_SB) by default
    ovm_report_message("pw_predictor_checker", $psprintf("%d porti_id", port_id));
    sb_aports[port_id % NUM_SB].write(trans);
endtask // transfer
```

The `pw_predictor_checker` class provides a built-in virtual method `transfer()` to handle different input and output data type. By default, input and output data type of the `transfer()` method are the same.

The code snippet above shows the default `transfer()` method. Actual transfer functions may be quite complex and dependent upon mirrored images of DUT state and multiple modes or configurations.

Step 5. End of test check and statistics

```verilog```
class xbus_demo_tb extends ovm_env;
    function void report();
        pw_sb.report_sb(1); // (chk_outstanding, rpt_outstanding)
    endfunction
    ...
```

SVF scoreboard provides methods to report the statistics of the scoreboard, such as how many data have been posted on the scoreboard, how many data have been checked, how many outstanding data are left on the scoreboard. The example above shows how the scoreboard reporting method is called in the `report()` phase of the simulation.

5.1. Advanced Usage Example: Posting Scoreboard Data with Timeout Events

The following set of code snippets show the user code to set up timeout events associated with the posting of transactions to `post_sb_data()`.

Step 1. Define a task to fork a thread to delay and trigger an `ovm_event` upon timeout

```verilog```
task delay_to_trigger_timeout(ovm_event ev_timer);
    fork
        begin
            ovm_report_info("", "In delay_to_trigger_timeout, prior to delay...");
            $sleep(3000); // a very simple fixed delay
            ev_timer.trigger();
            ovm_report_info("", "In delay_to_trigger_timeout, just triggered event...");
        end
    join_none
endtask // delay_to_trigger_timeout
```
Step 2. Declare `ovm_event` handle for use in attaching an `ovm_event` to a transaction as it is posted:

```cpp
ovm_event ev_timer;
```

Step 3. Create an `OVF_event` and post it to the Scoreboard:

The following code snippet shows how the monitor creates an `ovf_event` via `new` and posts it to the scoreboard. The example uses a cloned copy of the generated transaction and uses stream_id '0' in the call to `post_sb_data()`.

```cpp
// Clone transaction 't'
$cast(t_cpy,t.clone());
ev_timer = new;
```

```cpp
// post the transaction with stream_id=0 and ovm_event
pw_sb.post_sb_data(t_cpy,0,ev_timer);
```

```cpp
// call task to fork timeout thread for this transaction
delay_to_trigger_timeout(ev_timer);
```

When the timeout is set to less than the time it takes to transit the scoreboard, or when the packet is mistakenly dropped, allowing the timeout to trigger, the scoreboard will report an error similar to this:

```
# OVM_ERROR @ 1108: ovm_test_top.pwr_demo_sve0.pw_sb[] [] Timed out on event : for transaction:a:1 p:2 r:10 t: 10 [f6_b9_74_64_fc_cc_c9_b3_b4_fc] parity : 0x1e
```
Today's complex, multi-million gate SoC designs consume a lot of development time, and hence money, with every simulation, synthesis or place & route run. At the same time, the pressure to be profitable and early to the market shortens design cycles and leaves almost no margin for errors.

Simulation of large RTL-designs is computation and time intensive. Interfacing with other physical systems cannot be included easily into RTL-simulations and is therefore tested on rapid prototyping systems. Wouldn’t it be great if you could use your emulation platform as a hardware accelerator for your RTL-simulations and be able to quickly jump back and forth between the world of simulation and the world of emulation?

Gleichmann Electronics Research’s SEmulator® is such a bridge between the world of simulation and the world emulation.

SEmulator® is a synthetic word that combines the words "Simulator" and "Emulator". It describes the basic functionality of the SEmulator® very well. The SEmulator® provides bridging functionality between the domain of digital hardware simulation and the world of FPGA prototyping. Design blocks can easily be moved between these two domains.

SEmulation, or simulator controlled emulation, combines digital hardware simulation with emulation in a rapid prototyping system, allowing the step-by-step transfer of functional blocks from the simulator (software) into the FPGA (hardware) without leaving the simulation environment and without the need to recompile the complete design for every minor modification, thus shortening the development time.

Using dedicated hardware instead of a general purpose CPU to perform certain computation intensive tasks is known as hardware acceleration in computer science. In the context of RTL-simulation, hardware accelerators have been used in the ASIC world for a long time. They are expensive machines, most of the time without the possibility to interface with other systems that require real-time behavior.

What is smart about the SEmulator® is that it can reuse standard emulation hardware, like ARM’s “Microcontroller Prototyping System” (MPS), Altera’s Automotive Platform (PARIS), the Industrial Control Platform (Hpe®_IRP) recommended by Intel or even customer specific

emulation hardware. All these platforms have many interfaces to the outside world. An optional PCI Express X4 link can turn these emulation platforms into a hardware accelerator for RTL-simulation of complex SoC designs, speeding up simulation time by a factor of 100 and more. Of course this factor depends on the complexity of the design blocks that have been moved to hardware. For instance, if you just move a NAND-Gate to hardware, the communication overhead will even slow down your simulation, whereas for design blocks of 50,000 ASIC gate equivalents and more a significant speed-up can be observed.

Creative solution: reusing emulation hardware for hardware acceleration

In the simulation process, an additional step has to be made: the designer has to specify the design blocks that are intended to remain in software and the ones that are supposed to be moved into hardware. This is done within the Hpe®_desk software framework (Hpe is short for “Hardware Prototyping & Emulation”). The SEmulator® synthesis flow will automatically synthesize blocks that are not synthesized yet and store them in a cache. Finally a top level synthesis will connect the inputs and outputs of each block to an IO-Manager that converts PCI Express to stimuli and hardware responses to PCI Express.
Since the SEmulator® consists of standard emulation hardware for the most part it is available at a fraction of the cost of high-end hardware accelerators. For the first time, hardware acceleration becomes affordable even for FPGA designers. Historically FPGA designers did not bother too much about simulation. Reprogrammable logic had low gate count and compile times were relatively short, so “trial and error” was the method of choice. This is no longer true for today’s high-end FPGAs. With compile times of several hours, FPGAs designers are changing their methodology and spend more time simulating their designs.

Co-simulation with real hardware

Speeding-up simulation time is just one benefit. More importantly, the system allows co-simulation of real hardware blocks (e.g., Ethernet, display controllers) with an existing HDL simulation. This approach allows integration of real hardware early in the design cycle; even co-simulating hardware blocks for which no simulation model is available. A so-called clock acceleration mode allows changing from the clock provided by the simulator to a real-time emulation clock. Even when speeding up simulation with hardware acceleration, the maximum clock frequency of the clock generated by ModelSim SE ranges anywhere from a few to few hundred kHz. Also the simulator clock has no stable and defined frequency, whereas clock sources on emulation hardware are typically in the 10 to 100 MHz range. The clock acceleration mode is similar to “fast forward” on a VCR: you can either watch a movie or “fast forward”. In case of clock acceleration you can either observe signals in your simulation or you can use clock acceleration to “fast forward” parts of your simulation or to include interfaces that require real-time behavior. For instance in the benchmarks show in figure 4, clock acceleration was used to speed up memory accesses.

Commercial position of the SEmulator®

Let us illustrate this flow with the classic example used to demonstrate hardware acceleration: the iterative calculation of the set of complex numbers known as Mandelbrot set, named after the “father of fractal geometry” Benoît Mandelbrot. The design is relatively small using just 12,000 ASIC gate equivalents, memory and some DSP blocks. The results of the calculation are stored in a RAM block that can hold the contents of one picture with 256 x 256 pixels in 256 colors. Calculating and storing one picture in real time would require approximately 212 ms. A pure RTL simulation of these 212 ms required a little bit more than 4 minutes (251 sec) of computation time. The Mandelbrot set has become popular outside mathematics for its aesthetic appeal, which is demonstrated with a viewer.
RTL-Simulation results of the Mandelbrot calculation can be displayed in a viewer.

The top-level testbench is reading out the RAM content and is dumping it to a data file named vh.dat, which can be displayed with the viewer.

Let us assume that the designer is at a point where the design performing the Mandelbrot calculation is considered stable and now an image processing block has to be designed. Since the Mandelbrot block is stable it can be transferred to hardware to speed-up RTL-simulation and to be able to have it interact with interfaces that require real time behavior (e.g. a DVI-monitor). In a first step it has to be specified what parts of the design are supposed to be moved into hardware. In this example the complete top-level design is marked as hardware block, indicated by the red coloring in the Hpe®_desk design tree. The top-level testbench remains in software, indicated by black coloring. With just three mouse clicks this design can be moved to hardware:

- Click 1 runs the block-level synthesis, connects each block to an appropriate IO-Manager and generates an FPGA bitstream. The IO-Manager takes care of the communication between the device under test and the “SEmulator® Interface Box”, which is the physical bridge between host-PC and emulation hardware.
- Click 2 is downloading the bitstream generated with the first click to the specified FPGA.
- Click 3 is finally initializing and opening the ModelSim simulator.

With three mouse clicks a SEmulation bitstream can be generated and downloaded to an FPGA.

The first click is quite time consuming, so it makes sense only for relatively large and stable design blocks or design blocks that are supposed to interact with systems that require real-time behavior. Now the same simulation that required 251 seconds finished in just 104 seconds. This may not sound all that impressive, but keep in mind that the design block that was moved to hardware was relatively small.

Suppose the design block that is doing the Mandelbrot calculation has been used in many projects before and the confidence level that is has been well tested is very high. The only thing that has been modified is the memory block and it has to be verified that the calculation results are stored properly. Now clock acceleration can be used to fast forward the calculation process by calling a special function named hac_clk(). This function will use a real clock source located on the emulation hardware instead of the clock provided by the simulator. Below is an example for a clock generation process that is constantly switching between simulator clock and emulation clock.
clk_gen: process is
  variable num_clks: integer := 10000;
begin
  if (now > (ImageStartTime + 1 us)) and
  now < (ImageStartTime + cEnableMemReadTime - 1000 us) then
    hac_clk(0, num_clks, 2);
    wait for num_clks * 20 ns;
  else
    wait for 10 ns;
    clk <= '1';
    wait for 10 ns;
    clk <= '0';
    end if;
end process clk_gen;

Using clock acceleration further reduces simulation time from 251 seconds to just 1.7 sec—more than a factor 140!

With this speed-up, simulations that had to be run overnight can now run almost interactively in a few minutes. More importantly interfacing with systems that require real-time behavior can be included in the simulation. During clock acceleration signals cannot be observed. For instance in the screen shot below the clock signals holds its value (zero) until the clock acceleration is turned off again. If required these gaps could be filled with on chip logic analyzers provided by most FPGA vendors.

CONCLUSION

Today's multi-million gate designs force designers to spend more and more time for simulation and verification. Since FPGA compile times have become significant, even FPGA designers that got away with “trial and error” in the past cannot afford to skip simulation in their design flow. High-end hardware accelerators used in the ASIC world cannot be used in an FPGA flow, where tooling costs have to remain low.

The SEmulator® is a system that uses standard components and software like FPGA prototyping system, a personal computer and an RTL-simulator and turns them into a hardware accelerator. The “SEmulator® Interface Box” or in some cases simply a PCI Express cable can be used as the physical bridge between emulation hardware and the PC that hosts the simulator. In clock acceleration mode the interfaces of the prototyping system can be used to include real-time interaction with other systems in the simulation. The SEmulator® is available at a fraction of the cost of high-end hardware accelerators, making hardware acceleration affordable for an FPGA design flow.

For more information refer to www.semulator.de or contact marketing@ge-research.com

SEmulation reduces RTL-simulation time. The clock acceleration mode allows including real time behavior in the simulation, however signals cannot be observed in this mode.
Coding Concise SystemVerilog Assertions.
by Clifford E. Cummings, Sunburst Design, Inc.

ABSTRACT
The introduction of SystemVerilog Assertions (SVA) added the ability to perform immediate and concurrent assertions for both design and verification, but some engineers have complained about SVA verbosity or do not understand some of the better methodologies to take full advantage of SVA.

This article documents a valuable SVA trick to generate concise design assertions. The concise coding style detailed in this article can reduce concurrent SVA coding efforts by 50%-80% over conventional SVA coding techniques.

1. INTRODUCTION
Engineers who have taken SystemVerilog Assertion (SVA) training are typically excited about the capabilities of SVA, but later many largely abandoned the use of SVA due to the perceived verbose nature regarding the creation and implementation of SystemVerilog assertions. There is a simple trick that every design engineer should know to build concise SystemVerilog Assertions.

1.1 What is an assertion?
An assertion is basically a “statement of fact” or “claim of truth” made about a design by a design or verification engineer. If that claim can ever be proven false, then the assertion fails (the “claim” was false). Assertions essentially become active design comments.

2. CONCURRENT ASSERTIONS
The most valuable assertion style used in design and verification environments is the concurrent assertion. Concurrent assertions are little monitors that sit inside of a block of code to periodically sample and test signals and generate error messages if the assertion ever fails.

Concurrent assertions are typically sampled once per clock period at the end of the clock cycle, just before the next active clock edge.

Concurrent assertions require the assertion of a property, where a property is basically a design rule that should always be true. The simplest of concurrent assertions takes the form:

```
assert property ( property_definition );
```

**Example 1 - Simple property assertion**

This example property requires a sampling signal (typically a clock edge) that is explicitly listed in the property definition.

The property definition can also specify a condition under which the property is disabled, followed by either a Boolean expression that should always be true or a user-defined sequence of signals that should always be true as shown in Example 2.

```
assert property (
    @sample_signal
    disable iff ( expression ) // optional disable condition
    property_expression_or_sequence
);
```

**Example 2 - Simple property assertion with property definition details shown**

Each property can be declared individually and then separately asserted as shown in Example 3.

```
property p1;
    @sample_signal
    disable iff ( expression ) // optional disable condition
    property_expression_or_sequence
endproperty
assert property ( p1 ) optional_action_block ;
```

**Example 3 - Separate property definition with subsequent property assertion**

The separate property declarations can be grouped into a library of properties to be asserted on multiple designs.
3. CONCISE ASSERTION CODING STYLES

I have observed that most engineers like the idea of using concurrent assertions, and enthusiastically embrace them in training, but frequently design engineers use them sparingly or abandon them altogether. The reason for this abandonment seems to be the verbose nature of declaring individual properties and then being forced to assert the properties separately.

One useful technique to create concise and yet powerful assertions is to use simple macro definitions.

3.1 Macros with arguments

The 1995 Verilog Standard[2] specified that macros could be defined on multiple lines by adding the backslash (\) continuation character just before the newline character. At the point where the macro substitution takes place, all of the macro code is inserted with newlines but without the continuation characters.

The 1995 Verilog Standard also added the ability to pass arguments to macros and that the scope of the arguments extended up to the end of the macro definition.

Consider the concurrent assertion for a simple D-flip-flop shown in Example 4:

Example 4 - Assertion command with long descriptive label

This assertion contains several pieces that are likely to be repeated across multiple assertion definitions. The pieces that are likely to be repeated include:

- assert property - keywords to start the definition of an assertion.
- ( ... ) - place holder for the assertion.
- @(posedge clk) - sample signal for the concurrent assertion.
- disable iff (rst_n) - definition of when the assertion should become inactive.

ERROR_q_did_not_follow_d:
assert property (@(posedge clk) disable iff (!rst_n) (q==$past(d)));

The only part that is likely to be unique to the assertion in Example 4 is:

- q==$past(d) - actual assertion test.

We can replace the repetitive portions of the assertion with the incomplete macro definition, as shown in Example 5:

Example 5 - Incomplete assertion macro with commonly used assertion code

Then we can complete the macro by adding the ability to pass the actual assertion test code into the macro using the argument (arg) as shown in Example 6. Note: there can be no space between the macro name “assert_clk” and the open paren for the argument.

Example 6 - Completed assertion macro with argument passed to the macro

With this macro definition in place, it is now possible to re-code the assertion of Example 4 using the macro definition as shown Example 7.

Example 7 - Macro with argument used to declare concurrent assertion

After creating a few key macro definitions, the job of writing assertions becomes much more concise and much easier to do.
3.2 Measuring the efficiency of macro assertion coding styles

So how effective are the complex macro definitions with arguments discussed in section 3.1 compared to the concurrent assertion coding styles of section 2 that are typically used by many engineers? To measure the efficiency of different assertion coding styles, this section will examine a synchronous FIFO SVA example coded using: (1) separate properties and assertions, (2) combined properties and assertions, and (3) macros and assertions with arguments.

3.2.1 Synchronous FIFO assertion subset

Consider the example of a 16-deep, 1-clock synchronous FIFO design. Four sample assertions that could be applied to test the FIFO design with respect to correct operation when the FIFO is either asynchronously reset or full/near-full conditions include:

1. When the FIFO is reset, the FIFO empty flag should be set and the full flag, wptr (write pointer), rptr (read pointer) and cnt (word counter) should all be cleared.
2. If the word counter (cnt) is greater than 15, the FIFO is full.
3. If the word counter (cnt) is less than 16, the FIFO is not full.
4. If the word counter is 15 and there is a write operation without a simultaneous read operation, the FIFO should go full.

This subset of synchronous FIFO assertions will be coded three different ways: (1) using individual property declarations and separately asserting each property (shown in Section 3.2.2), (2) asserting each property without a separate property declaration (shown in Section 3.2.3), and (3) using assertion macros (shown in Section 3.2.4).

3.2.2 Separate properties and assertions

One of the first techniques typically shown in assertion books and taught to engineers in typical assertion training, is the declaration of separately named properties, followed by the assertion of the named properties. Although the technique has merits especially for verification teams that intend to construct a large set of reusable properties that can then be used by others, the technique is far too verbose for the average design engineer.

The FIFO assertion subset described in Section 3.2.1 is declared as a set of properties, each one separately asserted, as shown in Example 8.

```
// Asynchronous reset property:
property reset_rptr0_wptr0_empty1_full0_cnt0;
@ (posedge clk)
(rst_n |->
 (rptr==0 & wptr==0 & empty==1 & full==0 & cnt==0));
endproperty

// FIFO full condition properties:
property full_fifo_condition;
@ (posedge clk) disable iff (!rst_n)
(cnt>15 |-> full);
endproperty

property not_full_fifo_condition;
@ (posedge clk) disable iff (!rst_n)
(cnt<16 |-> !full);
endproperty

property fifo_should_go_full;
@ (posedge clk) disable iff (!rst_n)
(cnt==15 & write & !read |-} full);
endproperty

// Now assert the predefined FIFO properties. Asynchronous reset
// assertion:
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
assert property (reset_rptr0_wptr0_empty1_full0_cnt0);

// FIFO full condition assertions:
ERROR_FIFO_SHOULD_BE_FULL:
assert property (full_fifo_condition);

ERROR_FIFO_SHOULD_NOT_BE_FULL:
assert property (not_full_fifo_condition);

ERROR_FIFO_DID_NOT_GO_FULL:
assert property (fifo_should_go_full);
```

Example 8 - FIFO assertion subset declared as separate properties and assertions

The declaration and assertion of these properties requires 25 lines of code (blank lines omitted) and 804 characters. That is a lot of code and effort to monitor four potential error conditions, which is why design engineers quickly abandon this assertion coding style.

3.2.3 Combined properties and assertions

Another technique frequently shown to engineers in books and assertion training, is the declaration of asserted properties without separate declaration of named properties. Although this technique
does work, the technique is still too verbose for the average design engineer who might intend to construct design-specific assertions.

The FIFO assertion subset described in Section 3.2.1 is declared as a set of combined properties and assertions as shown in Example 9.

```plaintext
// Asynchronous reset assertion:
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
assert property (@(posedge clk)
    (rst_n -> (rptr==0 && wptr==0 && empty==1 && full==0 && cnt==0)));

FIFO full condition assertions:
ERROR_FIFO_SHOULD_BE_FULL:
assert property (@(posedge clk) disable iff (!rst_n)
    (cnt>15 |-> full));
ERROR_FIFO_SHOULD_NOT_BE_FULL:
assert property (@(posedge clk) disable iff (!rst_n)
    (cnt<16 |-> !full));
ERROR_FIFO_DID_NOT_GO_FULL:
assert property (@(posedge clk) disable iff (rst_n)
    (cnt==15 && write && !read |=> full));
```

Example 9 - FIFO assertion subset declared as combined properties and assertions

The declaration and assertion of these properties requires 13 lines of code (blank lines omitted) and 515 characters. Although not as verbose as the separate property declarations and assertions of Section 3.2.2, it is still a lot of code and effort to monitor four potential error conditions, which is why design engineers also quickly abandon this assertion coding style.

For most design engineers, these last two assertion coding styles are the only styles that engineers have been taught, so many design engineers abandon adding assertions altogether.

3.24 Macros and assertions

The technique that I encourage most design engineers to use is to define a couple of simple, yet powerful, macros that can reduce the coding effort required to add assertions to the typical design.

The FIFO assertion subset described in Section 3.2.1 is declared using a couple of simple macros as shown in the assertion macro definitions of Example 10.

```plaintext
// Assertion macro definitions:
`define assert clk(arg) 
    assert property (@(posedge clk) disable iff (!rst_n) arg)

`define assert_async_rst(arg) 
    assert property (@(posedge clk) arg)

Asynchronous reset assertion:
ERROR_FIFO_RESET_SHOULD_CAUSE_EMPTY1_FULL0_RPTR0_WPTR0_CNT0:
    `assert_async_rst(!rst_n |->
        (rptr==0 & wptr==0 & empty==1 & full==0 & cnt==0));

FIFO full condition assertions:
ERROR_FIFO_SHOULD_BE_FULL:
    `assert clk (cnt>15 |-> full);
ERROR_FIFO_SHOULD_NOT_BE_FULL:
    `assert clk (cnt<16 |-> !full);
ERROR_FIFO_DID_NOT_GO_FULL:
    `assert clk (cnt==15 & write & !read |=> full);
```

Example 10 - FIFO assertion subset declared and asserted using concise macro definitions

The declaration of the macros and use of the assertion-macros requires 13 lines of code (blank lines omitted) and 521 characters. The declaration of the assertions omitting the macro declarations requires 9 lines of code (blank lines omitted) and 365 characters. This is a reasonable effort to monitor four potential FIFO error conditions. This technique permits rapid definition of assertions that offer great value to the design engineer. I have found that design engineers are much more willing to adopt assertion based design capabilities when presented with this simple, yet powerful, assertion macro technique.
A good way to approach the use of assertion macros is to think of the assertion label as the comment that describes the condition if the assertion fails, thereby documenting the intent of the assertion, followed by the actual assertion test. All of the tedious overhead-code of the assertion has been collected into the macro definition itself.

3.3 Assertion coding benchmarks

So what is the effort required to code a reasonable set of assertions using the three techniques described in the preceding sections?

Reference [1] includes a set of 13 assertions that could reasonably be applied to a 1-clock synchronous FIFO design. The assertions are coded using the three techniques described in the preceding sections.

After coding the assertions using all three techniques, the code volume was measured as the number of lines of code required and characters used to code each set of assertions. The blank lines were omitted from the measurements. The assertion macro definitions were also omitted from the measurements under the assumption that as more macro-assertions are added to a design, the six lines of macro definition code would eventually become insignificant.

Table 1 - Assertion coding effort

<table>
<thead>
<tr>
<th>Style (blank lines omitted)</th>
<th>Lines of code</th>
<th>Additional lines of code (%)</th>
<th>Characters</th>
<th>Additional characters (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property/Assert Property</td>
<td>79</td>
<td>193%</td>
<td>2599</td>
<td>126%</td>
</tr>
<tr>
<td>Assert Property</td>
<td>40</td>
<td>48%</td>
<td>1707</td>
<td>48%</td>
</tr>
<tr>
<td>Macros</td>
<td>27</td>
<td>0%</td>
<td>1151</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table 2 - Assertion coding effort with labels omitted

<table>
<thead>
<tr>
<th>Style (with no labels and blank lines omitted)</th>
<th>Lines of code</th>
<th>Additional lines of code (%)</th>
<th>Characters</th>
<th>Additional characters (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Property/Assert Property</td>
<td>66</td>
<td>371%</td>
<td>2067</td>
<td>249%</td>
</tr>
<tr>
<td>Assert Property</td>
<td>27</td>
<td>93%</td>
<td>1175</td>
<td>98%</td>
</tr>
<tr>
<td>Macros</td>
<td>14</td>
<td>0%</td>
<td>593</td>
<td>0%</td>
</tr>
</tbody>
</table>

It can be seen from Table 2 that the coding effort required to add one of the traditional assertion coding techniques required approximately 100%-250% more characters than what was required to add the same assertions using the concise macro definitions.

Any of these SVA coding styles work well, but I have found that engineers are much more willing to add assertions to their designs if the effort required to add the assertions is reasonable. The concise macro definitions offer a much more attractive option over traditional SVA coding styles.

4. CONCLUSIONS

Design engineers frequently do not use SVA because they perceive that the effort to code SystemVerilog properties and assertions is too verbose for the potential debugging benefit derived from adding the assertions.

The concise `assert_clk` and `assert_async_reset` macros significantly reduce the effort required to add assertions to an RTL design. In the examples cited, the `assert_macros` were shown to reduce assertion coding efforts by 50%-80% over conventional SVA coding techniques.

The easier it is to add assertions to a design, the greater the likelihood that design engineers will embrace the usage of design assertions. Concise macro usage translates into greater acceptance of assertion deployment by design engineers.

If design engineers will use the SVA macros shown in Section 3.2.4, their designs will be significantly better tested than a design where assertions have been omitted and the design engineer will spend less “quality time” with the verification engineers!
5. REFERENCES


ABOUT THE AUTHOR

Cliff Cummings is President of Sunburst Design, Inc., a company that specializes in world-class Verilog, SystemVerilog and synthesis training.

Cliff has participated on every IEEE & Accellera Verilog, Verilog Synthesis, and SystemVerilog committee, and has presented more than 40 papers on Verilog & SystemVerilog related design, synthesis and verification techniques, including 17 that were voted “Best Paper.”

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site, or email: cliffc@sunburst-design.com
ABSTRACT

As product designs become more complex and there is less time to deliver these complex designs, the importance of delivering a first-time correct design and earlier hardware/software integration is becoming a priority in the product design cycle. One way to achieve the goal of a first-time correct hardware design is to use board level functional simulation. The Mentor Graphics Seamless® co-verification tool can be used in the board level simulation environment to perform hardware/software co-verification before the first physical prototype boards arrive in the lab. This not only provides earlier hardware/software integration testing, but also provides earlier and greater visibility of any problems that may arise during that integration.

This paper describes:

- the objectives of board level simulation and hardware/software co-verification
- a verification flow and verification environment using Seamless
- different verification methodologies to address specific verification challenges
- lessons learned
- benefits derived and trade offs from employing a board level functional simulation and hardware/software co-verification methodology using Seamless

OBJECTIVES

The objectives of board level functional simulation are to obtain a first-time correct design by verifying the connectivity of the design and the interaction of devices in the hardware design. The verification of the connectivity of the design can help to determine whether the schematic has been entered correctly and whether there will be functional connectivity problems between the devices in the schematic. When you verify the interaction of the devices in the hardware design, the compatibility of the devices in the design can be assured. For example, you can verify that a processor interacts correctly with a bus bridge controller.

The objectives of hardware/software co-verification are to integrate software into the board level simulation effort, which not only provides stimulus for the board level functional simulation, but also allows the software to be verified earlier in the product life cycle. Specifically, software boot code and low level drivers can be verified at this point of the design cycle. Also, various portions of device diagnostic software can be tested. The verification of portions of software at this stage of the design cycle allows the integration of software into the first hardware prototypes to occur quite quickly.

VERIFICATION FLOW

The verification flow for board level functional simulations and hardware/software co-verification includes design verification, test bench development, and supporting tasks. These activities are driven by the requirements of the design, as shown in Figure 1.

After the requirements of the design are documented, the schematic of the design can be captured using a schematic capture tool such as Mentor Graphic’s Design Architect®. In parallel with this activity, development of the board level verification test bench should be started. Although the overall test bench for board level functional simulations is the captured schematic, other items need to be developed for the test bench. The items include creating and/or adjusting models that can be used for simulation, and creating the stimulus and checkers to execute and verify the simulation.

In addition, the supporting tasks for board level functional simulation need to be developed and completed. The tasks include documenting the test plan and test cases, creating a working environment, and obtaining device models. These supporting activities should be started when the requirements of the design are available.
The test plan document provides the strategy of how each part of the design will be verified and indicates the type of models to be used for the different circumstances of a specific simulation. The test case document indicates the test cases to be executed in the simulation and provides a record of the test cases that pass and fail.

The working environment is the verification environment for the simulations. The environment provides the supporting scripts to extract the schematic netlist and/or customize the netlist. It provides the scripts that can check the results of the simulation and provides automated regression testing for any design changes. The working environment also archives the simulation environment, supporting scripts needed to reproduce the simulations, and the simulation results.

Model procurement is the acquisition of the models for the design. There are different types of models that can be obtained at different levels of abstraction. These types of models include encrypted and non-encrypted RTL device models, behavioral device models, bus functional models of the device’s interfaces, and high-level language models that work in the simulation environment.

Some of these models can be obtained from the vendors of the devices that are used in the design. Also, some of these models can be obtained from model vendors such as Denali for memory models or Synopsys for LMC Smart Models. For Seamless simulations, a Seamless model is provided by Mentor Graphics. If Mentor does not have a Seamless model for a processor, contact Mentor to determine whether a Seamless model can be obtained for that processor.

If a model cannot be obtained for a device in the design, an in-house model may need to be developed. Another alternative is to create a bus functional model for the device’s interfaces.

As mentioned earlier in the paper, during test bench development, some of the models may need to be adjusted to fit into the simulation environment. One way to do this is to write wrappers around the model. Another adjustment to the models may be to place stimulus generators and/or automated checkers within a model wrapper to provide specific stimulus for a simulation or to check for specific desired results of a test case performed in a simulation.

When the schematic is ready for simulation and the working environment is created, the netlist of the schematic can be extracted for board level functional verification by using the Mentor Graphics HDLWrite™ tool. The HDLWrite tool creates a Hardware Description Language (HDL) extracted netlist version of the schematic in an HDL such as VHDL or Verilog.

With an HDL netlist version of the extracted schematic, the netlist can be customized to specify the portions of the design to be simulated. Because of the large number of cycles needed to execute a simulation, it is highly recommended that you divide the simulation into tasks that verify specific portions of the schematic at one time. Another task of netlist customization is to remove items that do not add any benefit to functional simulation such as series resistors and decoupling capacitors. It is highly recommended that you develop scripts to make the netlist customization step repeatable and easy to perform. The netlist customization step should be driven by the requirements of the specific test case to be executed in the simulations.

During the compilation step, the customized netlist of the schematic, models used for simulation, and the items in the simulation test bench are compiled for the simulation. The Seamless model is also compiled in this step. In this step, any compiler options for the simulation are specified.

During the simulation step, one of Mentor Graphics ModelSim® or Questa® simulators can be used as the simulation tool to perform the simulations. (For the rest of this paper, the simulation tool that will be referred to is ModelSim). Mentor Graphics Seamless CVE tool is used to provide the Seamless model and interface to the software executables needed for hardware/software co-verification.
The validation of the results indicates whether a simulation test passed or failed. If the simulation fails, inspection of previous steps in the verification flow is required to find out why the simulation failed; appropriate corrective action will need to be taken. Scripts should be written to validate the results of the simulation. These scripts should be expanded to provide regression testing for any future design changes that require verification using functional simulation and/or hardware/software co-verification.

VERIFICATION ENVIRONMENT

The verification environment for functional simulation and hardware/software co-verification using Seamless is a part of the overall product development environment, as shown in Figure 2. While the schematic is in Place and Route to create a Printed Circuit Board (PCB) for the Physical Prototype, a Virtual Prototype of the schematic is created for simulation where software can be executed using Seamless.

The verification environment comprises two distinct activities. The first activity is to convert the schematic into a Hardware Virtual Prototype to be used for simulation. The second is to use the Hardware Virtual Prototype to execute the board level functional simulations and hardware/software co-verification.

The creation of the Hardware Virtual Prototype requires a netlist of the schematic generated in Design Architect to be extracted using Mentor Graphics HDLWrite. Figure 3 shows the steps to create a VHDL extracted netlist to be the Hardware Virtual Prototype from the schematic.

To accommodate any future updates to the schematic, this step should be automated using scripts. Also, any options to be applied for the schematic using HDLWrite should be archived for repeatability on later versions of the schematic.

After the Hardware Virtual Prototype is created, the prototype can be used for functional simulation and hardware/software co-verification. Other inputs into the simulation include the various models that are needed to support the simulation test cases and the cross-compiled software executable that are used for hardware/software co-verification, as shown in Figure 4.

Figure 4 shows the VHDL Hardware Prototype of the schematic being used with the Seamless model to create a Virtual Prototype that the cross-compiled software executable can be verified on. In addition, the figure shows the various models that support the simulations being added to the Virtual Prototype.
An example of the type of simulation models that are used for various devices in the simulation environment is shown in Figure 5. The microprocessor is represented by a Seamless model. A Denali memory model is used to model the memory device. The bus buffer device is modeled by a Synopsys LMC Smart Model. Actual design RTL code is used to model (and verify) the PLD and FPGA that are used in the design. A vendor-supplied model is used to model a third-party device. For the other device in the design that does not have a vendor-supplied model for it, a locally developed model was used to model it for the simulations.

Figure 5: Example of Simulation Models

Although the microprocessor in Figure 5 is represented by a Seamless model, the Seamless model is actually comprised of two components: the Instruction Set Simulator component and the RTL/Behavioral model component. The Instruction Set Simulator models the part of the device that cross-compiled software executes on. The Instruction Set Simulator is implemented as a high-level software model. The RTL/Behavioral component models the parts of the microprocessor device that are not modeled by the Instruction Set Simulator (such as the microprocessor’s peripherals and memory controllers).

When the Instruction Set Simulator is implemented as a high-level software model, the simulation is faster than the Instruction Set Simulator modeled as RTL. Figure 6 shows the relationship of the two components of the Seamless model and how they interact.

Figure 6: Seamless Model and Tools

Seamless provides the X-Ray debugger which interacts with the Instruction Set Simulator to provide visibility of the cross-compiled software executable. The ModelSim simulator provides visibility of the hardware virtual prototype of the design, which consists of a VHDL version of the schematic, the in-house created RTL devices, device models from device vendors, and any non-optimized memory used in the design. The Seamless CVE tool provides access to the RTL and Behavioral components of the Seamless model along with access to the optimized memory used in simulation.

With non-optimized memory, Seamless uses ModelSim to access the memory models, which creates a hardware access that directs memory accesses through the ModelSim simulator. With optimized memory, Seamless does not use the ModelSim simulator when a memory device is accessed. Instead Seamless accesses the memory virtually, which provides a faster simulation. This is especially useful for software instruction accesses. In both the optimized and the non-optimized memory options, any contents written into the memory device during a simulation are stored in the memory device model for future retrieval.

Figure 7 shows an example of the execution of Seamless CVE being with ModelSim and the X-Ray debugger on a Red Hat Linux system. The Figure shows the different windows that are involved in using Seamless for hardware/software co-verification and how they relate to each other.
In Figure 7, the ModelSim window is in the background with the Seamless CVE control window shown on top of the ModelSim window in the lower left corner. In the upper right corner window, the X-Ray debugger shows the current location of the software code with the instructions displayed in assembly (there is an option to display code in C or C++). In the lower right corner window, a dump of memory at a specific location is displayed. The upper left corner window displays the contents of the registers for the processor.

VERIFICATION METHODOLOGIES

This section describes some verification methodologies that address specific verification challenges. In particular, the topics are accessing memory devices, in-house developed RTL devices, series resistors, automation, validation of results, regression testing, and archiving.

When you verify access to memory devices, first verify the access to the memory device as a non-optimized memory device to ensure that the device is connected correctly in the schematic and that the processor is correctly set up to access the device. After this verification, change the memory device to be an optimized memory device so that any accesses to that memory device will not go through ModelSim. This results in faster simulations.

If an internally developed FPGA or PLD is used in the design, use the hardware virtual prototype created from the schematic along with the RTL code of the devices to verify that specific parts of the device are correctly connected in the schematic. For hardware/software co-verification, Seamless can be used to verify that the software driver is correctly written for these devices. These verification activities can then form part of the RTL device verification strategy.

Series resistors can be removed during the netlist customization step. The scripts can identify the series resistors and remove them from the extracted HDL netlist. Another way to remove the resistors is during the netlist extraction stage; set a property on the resistors and have HDLWrite short the resistor out of the design. One item for a future improvement with this methodology is to have a series resistor model created that a simulator can handle as a series resistor.

One way to reduce the time needed to complete simulations is to plan for full automation when you create the verification environment. To facilitate the automation, create scripts and templates that are easy to modify and can be reused for future simulation activities. Also, create a strategy at the outset with the types of scripts that are needed to accomplish specific tasks. As a result, the scripts can be designed within a context of a system rather than a patchwork of scripts put together without planning or foresight. A planned system of well-defined scripts provides a higher degree of success in reusing parts of the verification environment in future projects.

Although the ModelSim simulator provides a GUI window that can be used to debug the simulations and validate results, a number of text files that are produced by the various tools in the simulation environment can be used to verify results. For example, the Seamless tool provides journal and log files that can capture results from memory accesses and other events that occur within the Seamless microprocessor model. The Denali memory models also provide log files which record access to their devices. When you create in-house models, send interesting events (such as errors or notes) to log files for inspection at the end of simulation.

When a simulation environment has automation built in and various log files are checked to validate results, regression testing scripts can easily be constructed for simulation and hardware/software co-verification to verify any changes in the design. Automated regression testing allows simulations to be sent off to a compute farm to be executed. If the regression testing fails, an inspection of the log files produced by the simulations can be performed to determine the cause of the failure.
The simulation environment should be archived, along with any simulation results and regression runs. This allows past simulations to be inspected if any regression testing fails. It also provides a place for reusable components of the simulation environment to be stored and improved on for future projects.

LESSONS LEARNED

When you use Seamless, you need to know which parts of the processor model will be modeled as the Instruction Set Simulator and which parts will be modeled as either the device’s RTL code or a behavioral representation of the device. In this way, a simulation test case can be designed to take advantage of how the processor model is constructed. Conversely, if any problems occur in simulations with a Seamless model, knowing how each part of the device is modeled may help identify the source of the problem.

Usually, when a Seamless model is chosen for a processor, not all of the processor’s functionality is used in the design. Before you use the model, work with Mentor Graphics to ensure that all of the functionality that the design will use the processor for has been tested for that model. In this way, any problems with the model are known and any future fixes for the model can be planned in the verification schedule.

The simulation environment should be created for easy reuse in future projects. The effort to create future simulation environments is reduced. Also automate the environment as much as possible to facilitate regressions testing for any fixes in the design to address problems identified during simulation.

When you create any in-house models for the simulation environment, create a test bench for the model to verify the model before the model is used for simulations. It is easier to debug an in-house model with its own test bench than in the board level simulation test bench.

Get the software team buy-in and involvement early. The software team can plan the co-verification activity within their schedule. You need to understand the software design schedule and to ensure that the milestones for the simulation schedule match the software schedule. Also determine the dependency of deliverable items so that they match the schedule. For instance, it is important for software to know when the hardware virtual prototype will be ready for co-verification. Hardware should know when a device driver will be ready to test the connectivity of the schematic to that device.

When you develop the board level functional simulation and archival environments, verify they work with the software environments. Software can then be easily brought into the simulation environment and the software team can easily use the Seamless co-verification tools.

Seamless is best introduced when you use an entirely new processor for a new design. The benefit of using Seamless is more obvious in this type of scenario because the amount of new software code to bring up the processor would be quite high. Also if the design has a number of new devices that require drivers to be written for them, the amount of code that needs to be verified in this situation is quite a bit higher.

BENEFITS DERIVED AND TRADEOFFS

A number of benefits are derived by performing hardware functional simulation and hardware/software co-verification using Seamless in the product development flow. These benefits include the ability to start hardware/software integration earlier in the design cycle which results in greater visibility of the integration. This integration also allows software debugging to occur earlier in the design cycle. Any design defects that are identified at this stage are less expensive and easier to fix as opposed to fixing them with the physical hardware prototypes.

Another benefit is that this methodology allows critical circuitry to be verified before the first physical prototypes arrive in the lab. This reduces the quality risk of the first physical prototypes and generally results in first-time correct designs.

With the reduced debug time in the lab on the first physical prototypes, these boards can be delivered to downstream groups with working software earlier in the design cycle. These downstream activities can take place sooner with higher quality hardware.

The major trade off in employing this methodology is that instead of spending time debugging in the back end of the design cycle, more up-front work is required to create the simulation environment and execute the simulations and hardware/software co-verification. This is especially true if this methodology is employed for the first-time. There is a high ramp-up time to learn the technology.

To help mitigate the up-front work for future projects, it is recommended that the verification environment is designed for easy reuse for future projects. It is also recommended that the verification environment is documented and archived so that other teams can easily reuse it.
Device vendors can help make this methodology easier to employ by increasing the number of models for their devices. They can also create models that are flexible by providing specific interfaces of the model to be operational depending on the simulation test case. Another area in which device vendors can assist is to provide high-level software models of their devices for simulation and hardware/software co-verification. Flexible models and high-level software models will lower the number of cycles required to simulate the device and provide more options for simulation test cases.

CONCLUSION

Board level functional simulation and hardware/software co-verification using Seamless methodology provide an opportunity to have first-time correct designs and to start the integration of hardware and software earlier in the product design cycle. This results in discovering design defects in hardware, software, and hardware/software integration earlier in the product development cycle and at a higher level of visibility. The fixes to these problems can be implemented at a lower cost. By finding these problems before the first hardware physical prototypes arrive, the debug cycle of these prototypes becomes shorter and downstream groups can receive these prototypes with working software earlier in the design cycle. The trade off for these benefits is that more up-front work is required to create the verification environment and execute the verification flow, as outlined in this paper. By making the verification environment easy to reuse, this up-front work can be reduced for future projects. Industry can also contribute to reduce the verification effort required with the suggestions that have been presented in this paper.