SYNOPSYS®
Accelerating Innovation

Quarterly newsletter for verification engineers

Issue 1 | 2013

Advanced Verification Bulletin

About This Issue

Welcome to the Advanced Verification Bulletin!

With every leap in design complexity, verification grows in importance. Consequently, the role of the verification engineer becomes more visible and grows more difficult. Greater access to the newest trends and thoughts in advanced verification can play a major part in aiding the verification community

To this end, we are pleased to present this inaugural edition of the Advanced Verification Bulletin (AVB). The goal of the AVB is to provide valuable practice, tool, and trend insights to you, the verification professional, so that you will gain a greater understanding of best practices and upcoming trends in Synopsys Verification.

Inside this issue, you'll find:

- A guest article from Warren Stapleton of AMD, who shares his observations upon verification practices & methodologies …
- An 'Inside Synopsys' view of emerging trends in debug, emulation, functional verification, and VIP …
- An update on upcoming events of interest to the verification community

We hope that you will find this issue of the AVB useful and timely!

Regards,

The Advanced Verification Bulletin Team

We welcome your comments, suggestions, and topic ideas, so feel free to send us your feedback at **avb@synopsys.com**.

Verification Integration in the SoC World

Warren Stapleton, Senior Fellow, AMD, shares his experiences and advice on implementing best practices in verifying today's SoCs

System-on-Chip (SoC) is hardly a new topic. Our industry has been designing them, verifying them, and manufacturing them for several years now. Even though we have completed our shift to an SoC design methodology, we have yet to master its verification challenges.

I work with AMD's verification methodology team on development of the long term strategy related to the verification arena. The team is responsible for developing commonly used verification IP and a swath of tools and technologies that include workspace management, SoC logic-design construction, IP-to-SoC delivery mechanisms, IP metadata systems, testbench frameworks and techniques, regression systems, and metrics. The work requires following a continuous improvement process and leads to decisions regarding vendor tool selection, driving the make versus buy decisions, and contributing to the architecture of many of the internal tools, processes, and testbenches.

At our size and level of complexity, verification becomes a game of trade-offs. The AMD Verification team is constantly determining how to apply the right mix of some new, but mostly existing methodologies to achieve the goal of building high quality products within the R&D budget. Increasingly, design-architecture is also playing a role in simplifying the verification problem and providing us with designs that are easier to verify in a hierarchical fashion.

In This Issue

However, we aren't only dealing with new design architecture. We have the challenge of verifying chips that combine our newer IP and architecture with legacy IP blocks. Although IP integration would normally be classified as a design issue, at AMD, many aspects of the challenge of integrating IP gets tackled by the verification team, and consumes a significant amount of its our time.

The EDA industry has done a good job of developing and improving point tools to help us manage our complexity. However, to manage the challenges of complex SoC development, we need a greater focus on the integration of the solutions themselves. And it's not just EDA tools that need to be integrated; we need to attack methodology integration and system-level tool integration.

All of this points to the need to nail down three things: intelligent design and verification practices, an integrated verification methodology, and continued development of industry-wide design standards.

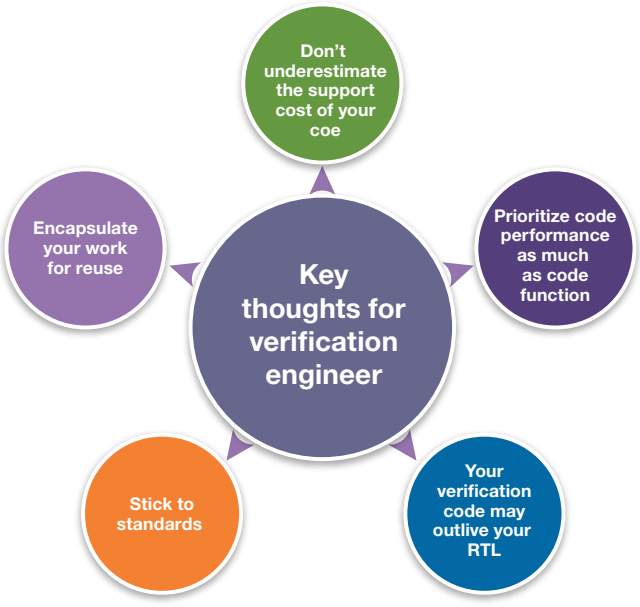
Designing for Integration

Every time a design or verification engineer writes a line of code, it comes with a cost. Not just the cost of their time to write and debug that code, but also the large cost of maintaining the code over timespans that go beyond the original developer's participation or initial intended application.

Today, when counting that cost, that engineer needs to understand that many of those lines of code will one day be reused.

If a designer's RTL code will become a functional block, it will be integrated with many other blocks on many different chips. It may even get sold as IP to a third party. It may also get integrated into a larger IP portfolio if the designer's company gets acquired one day. For the ease of integration, each one of these blocks needs to be encapsulated properly and adherent to industry standards.

It's just as important to understand that verification code will also be reused. When I started my career, we approached the problem as if our verification testbenches were throw-away work once a chip was complete. Today, verification IP may live longer than its targeted RTL. For example, consider the case of a TCP-IP core; the RTL for the core might be rewritten several times to optimize for power, area, or performance. However, TCP-IP functionality is set by a stable standard and does not change significantly over time. This example is repeated across the industry, so be aware that your verification IP will likely be reused again and again.



In the case of reusable verification collateral, it's very important to make sure that your code follows industry standard approaches and takes advantage of the industry standard frameworks. An equally important point is to make sure your code is reasonably optimized for simulation. If your code runs 25% slower than it possibly could, it may not make a difference on one single block, but as you start putting more and more of those blocks together and simulation slowdowns occur across the whole system, a single slow piece of the environment can become an expensive bottleneck for the entire project.

Integrating Verification Methodology

At AMD, we set a high bar for the quality of our chips; to reach our goals, we perform extensive verification. For us, simulation is king—it is our primary tool and technique. We use formal methods to solve focused problems and turn to emulation and virtual prototyping techniques to address tasks that simulation cannot: such as booting operating systems.

This methodology is constantly evolving as our design challenges evolve. Significant effort goes into transitioning from one technique or toolset to another; we benefit most when we adopt new techniques only after their value has clearly been demonstrated.

However, as our simulation tasks grow, we have to introduce different approaches to verification. Solving the verification problem hierarchically is a common approach, but leads to new problems related to integrating various IP, testbenches, and common libraries at the SoC level. While not a traditional verification issue, the integration problem is being tackled by the verification methodology team and is consuming a lot of the available engineering time.

I expect the integration problem to be the next big area of development for the EDA vendors. While some methodologies are being built at the moment to help with this,

like UVM, IP-XACT and some exploration into mixed-language frameworks, they don't all work together seamlessly yet. Secondly, many new tools are point tools and don't integrated seamlessly into the main tool suite. This limitation means they require complex flow development to make them deployable.

For example, at the lowest level, System Verilog, UVM and constraint-based random environments are state-of-the-art solutions. But, as larger and larger pieces of the design are combined, SystemVerilog is not necessarily the right language to use for higher abstraction levels. We would desire to write reference models in C++/SystemC, and use those reference models in the low-level System-Verilog/UVM environments. Although there are some techniques for moving design work between these levels of abstraction, there isn't a clear standardized methodology on how to move our verification work across them today, so we improvise by creating customized solutions.

I am participating in an Accellera effort to develop a more standard approach to combining verification collateral which has been developed in different frameworks.

The Need for Design Standards

We have quite a collection of IP at AMD. We have CPU cores, graphics cores, high performance fabrics, and many others. We have verification IP in the form of Simulation, Emulation, FPGA and System Verilog testbenches and sophisticated virtual prototyping platforms.

Some of these were developed internally, some came from companies we acquired and others come from third parties that are developing leading IP. Even the internally-developed IP is created by teams that might be spread across the globe, with different expertise, different native languages and different approaches to design.

With this wide dissemination of our knowledge base, inconsistency seeps in. As happens frequently, an engineer devises a creative way to do something that doesn't exactly match a standard approach; as a solution to solve a localized problem.

This creeping inconsistency is the bane of every verification team. We have, over the years, spent a lot of time, compute power and, ultimately, money working on the integration problem.

To combat this with newer designs, we have developed a standards-driven internal design approach. For our verification IP, we have standardized on a System-Verilog and UVM-based methodology for simulation and built tools around the continually improving IP-XACT standard to ease our integration issues

But, we encourage the industry to continue developing standards to address the propagation of information between separate teams and companies, not only for the traditional IP design area, but related activities like design libraries, verification IP, simulation models, and key design related flows (like DFT and power flows). With these, we will be closer to a plug-and-play IP environment, where we can truly directly re-use parts of our own designs or those from third parties.

In Conclusion

SoC design is an increasingly distributed endeavor. With teams dispersed across the planet, IP from a variety of sources, and flows built from lots of point tools, verification will continue to be a challenge for us in the coming years.

To meet this challenge, we as an industry must focus on two words: standards and integration. Only by doing so will we manage the verification complexity and challenges that we will face in future generations of IC design.

About the Author

Warren Stapleton is a Senior Fellow in AMD's verification methodology team, where he is responsible for the development of AMD's long term verification-related strategy.

Previous to AMD, he has held engineering and management positions at Montalvo Systems, Azul Systems, Redback Networks, Siara Systems, Nexgen, and Austek Microsystems.

He earned degrees in both Electrical Engineering and Mathematics from the University of Sydney.

Empowering Diagnostics

Internally and externally developed verification IP is being leveraged across an increasing number of projects. This increased leverage makes coding testbenches for high performance and efficiency absolutely crucial. Similarly integrating externally developed IPs creates difficult-to-debug scenarios that can vastly diminish the efficiency that was supposed to be gained from leveraging external IP. Ease of integration, debug, and deployment is a fundamental goal at Synopsys both in the development of our VIP and in helping our customers develop and deploy their own.

Diagnostic tools are one of the many ways in which we empower our customers to develop optimized testbenches and integrate IP quickly and painlessly.

Rebecca Lipon
Product Marketing Manager for Functional Verification

VCS Diagnostic Tool Suite: Reducing Issue-Resolution Turn-Around-Time to Improve Verification Efficiency

Large-scale designs regularly encounter issues: functional bugs within the design as well as compute resource allocation, tool, and IP integration issues. VCS ships a suite of tools designed to assist in the resolution of these issues from tracing timescale and file inclusion discrepancies to automating log files for easier debug. Most unique in this suite of tools is the robust VCS Unified Profiler. In this article we will review these new options with particular focus on our integrated time, memory, and constraint profiler. This tools and most especially the Unified Profiler allow advanced verification users to tune their design for optimal speed and memory efficiency.

Optimizing Simulation Performance

Simulation performance degradation occurs for many reasons: coding style as well as indiscriminate usage of debug dumping, coverage collection, assertion messaging, and even advanced methodologies verbosity settings. Whenever a simulator is writing out more data than absolutely necessary, performance will be slower than optimal. Script reviews combined with common sense can assist with reducing inefficiencies, but until recently tuning for performance has been a bit of a black art—verification engineers have based their analyses of the likely culprit for a slowdown on the number of messages being dumped, the size of a memory, the depth of an array, etc. There have not been tools for engineers to specifically know which modules, components, tasks, entities, or constructs are consuming the most time and memory in their designs and how to improve their code.

The VCS Unified Profiler provides a consistent way to analyze time and memory consumption of code written in Verilog, VHDL, SystemC , PLI, DPI, and DirectC. The Unified Profiler also allows you to see dynamic testbench constructs, coverage, assertion, and constraint data to assess their resource utilization. The Profiler provides both a time and memory view, and can generate reports in HTML hyperlinked to the source code for high efficiency root cause analysis of memory leaks and coding missteps. The Profiler can also generate text reports for integration into custom applications or reports.

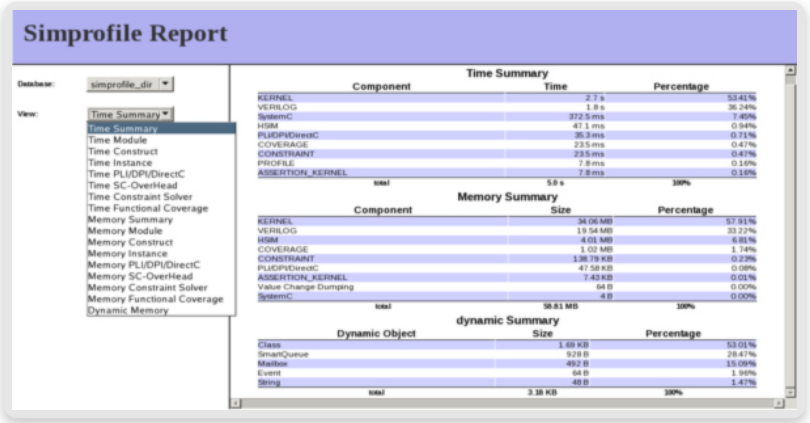


Figure 1: Unified profiler summary view

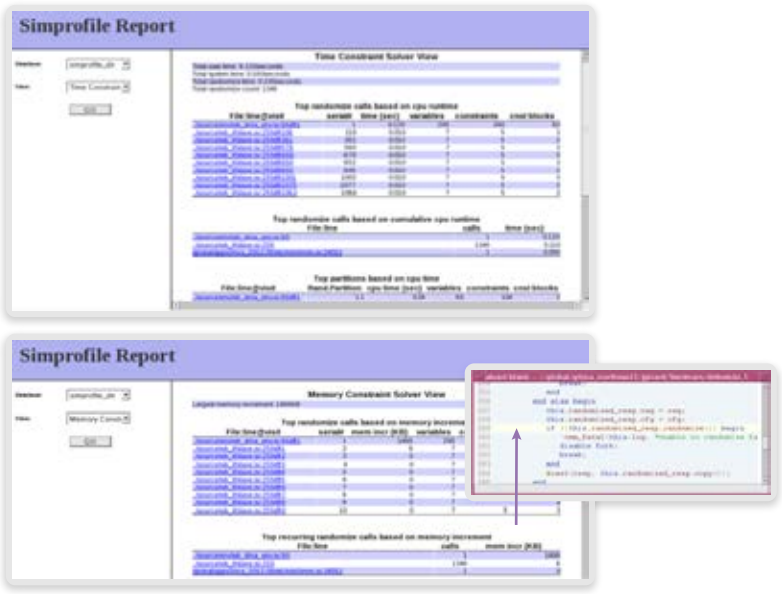


Figure 2: Constraint Solver View time and memory

The Unified Profiler is also fully integrated with VCS' constraint solver engine. The Profiler provides Time and Memory views that allow engineers to see in detail which calls to the `randomize()` method are using the most CPU time or the most machine memory. With this information verification engineers can consider revising constraints on random variables to use fewer of these resources.

Diagnostics: More Than Profiling

The VCS Diagnostic Tool Suite is not limited to the Unified Profiler Tool. The "-reportstats" utility exists for extracting simulation statistics allowing users to more easily determine to which regression pool they should assign simulation jobs. The Crash Context utility gives users data about compile, runtime, and out of memory crashes so users can more easily understand the issue that lead to the crash and get the tool up and running again. Diagnostics can provide detail on design binding issues that might have occurred when identically named ENTITY and ARCHITECTURE definitions exist in different libraries within a VHDL design and the content differs between those libraries. Bindings for v2k configurations can also be explored using this utility. Diagnostics can also be used to help resolve Verilog timescale issues.

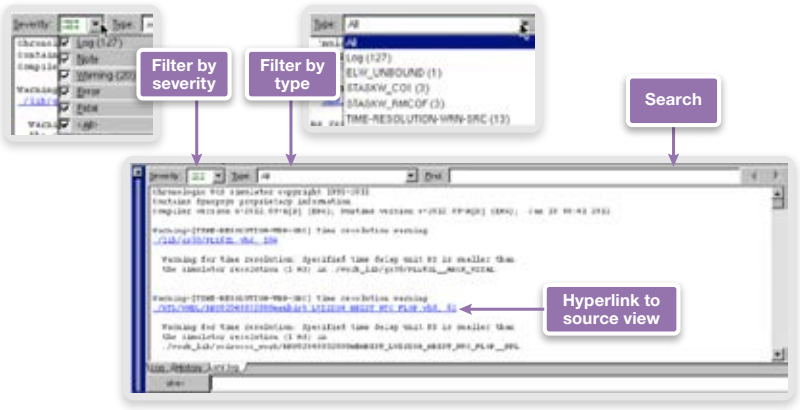


Figure 3: DVE smart logging

In Verilog if a timescale is not set for every compilation unit, it will be propagated from the previous files' definition. Tracing these timescales can be particularly onerous when IP integration occurs and an unknown timescale from the IP block accidentally is propagated to other regions of the design. Being able to quickly extract which modules set which timescales in what order allows users to quickly resolve these resolution issues and get back to the business of finding and fixing the real bugs.

VCS has also further extended the Discovery Visualization Environment (DVE) to help visually explore error messages and diagnostic feedback. DVE provides log analysis for each line in a log file, hyperlinks log occurrences to source files, highlights keywords such as "Error", "Warning", etc. in different colors to assist in easy tracing of design issues, and now displays the selected message within a blue rectangle so users know the context of the message they are tracing.

Summary

As design size and methodology complexity increases, improper resource allocation or bad coding styles can have a significant negative effect on engineering teams' efficiency. IP is increasingly being used across multiple organizations, projects and scopes, making utilities that help in the integration and resolution of the inevitable compilation and performance issues involved in bringing up these blocks imperative. VCS strives to give its users the best utilities to tune their code for performance, trace and resolve issues, and properly allocate compute resources so they can get the most out of the simulator. We are continuing to invest in these tools with improved utilities around race detection, testbench, constraint, assertion, and design debugging planned for our upcoming release. Here at Synopsys we are increasing the intelligence and automation of error resolution to help our users meet their intense schedule pressure.

Emulation Systems: An Essential Verification Element

We all recognize that with each succeeding generation of semiconductor technology, the number of processors and amount of embedded software in the SoCs is doubling. These changes are magnifying the problems of assuring the SoC meets its design specifications, and that the specifications meet the requirements when the SoC is employed in real world environments.

Emulation systems have been around, and evolving, for over 20 years. The level of SoC complexity has reached the point where emulation systems are now an essential element in the verification process. The current ZeBu product, a high performance, transaction-based SoC emulation system (now a part of the Synopsys verification portfolio) is the fourth generation, and reflects the latest technology employing the most advanced architecture and verification methodologies for accelerated debug of these ever increasingly complex SoCs.

We look forward to helping you to use ZeBu more effectively via the ideas and best practices that we'll explore via this issue (and future issue) of this bulletin!

Ralph Zak
Marketing Specialist for Emulation

Best Practices in SoC Emulation—Guidelines for Maximizing Value

What Has Changed in Emulation Usage Models

For many years, the principal emulation system use model was to take simulation-debugged RTL and map it as "early silicon" in reprogrammable hardware, and then operate it with real software while connected to a real physical environment. The goal was to gain confidence that the SoC would actually work as intended before committing to silicon. This verification methodology is referred to as in-circuit emulation, or ICE. In ICE, with the emulator running much slower than the connected physical environment, each system-level interface typically requires a data buffering mechanism to match the emulation system to the environment. In such environments with design specific hardware configurations comprising the verification environment, the emulation system access is essentially restricted to a single project at a time.

Maximizing the value of today's emulation systems requires taking different approaches than those of the past- namely: the use of virtual test environments and optimization of verification flow via a better mix of verification methods.

Virtual Test Environments Simplify the Use Model on Today's Complex SoCs and Increase Accessibility

There has been a large shift from ICE to transaction-based accelerated verification in which the emulated DUT interacts at very high speeds with a virtual environment. The key driver for this is the ever increasing number of external interfaces on SoCs- a tablet, wireless phone, or digital TV SoC may have >20 external connections, running the gamut of peripheral and communications protocols

The implementation of a transaction-based verification methodology provides many benefits over an in-circuit emulation methodology. The entire design is contained within its hardware and its associated PC: no target board is required, nor external cabling, level shifters or speed-adapters. Instead, the external environment is modeled as a group of transactor models for each aspect of the SoC interface; e.g. PCIe, USB, keypad, LCD display and camera sensors. The front end of each transactor that communicates at a high-level of abstraction with each peripheral is modeled in C on the PC (Figure 1).

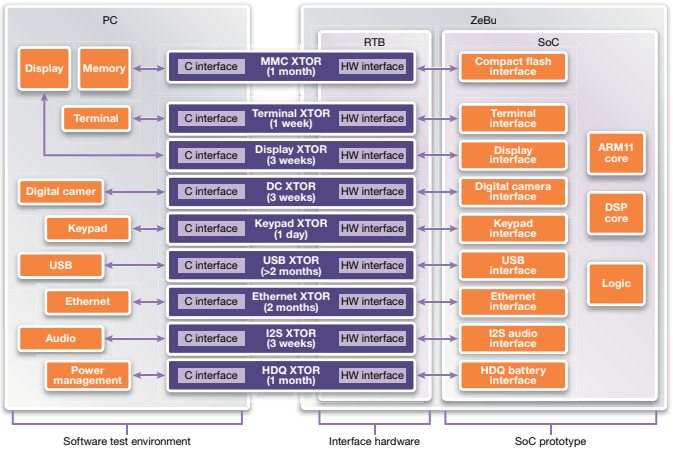


Figure 1: The front end of each transactor that communicates at a high-level of abstraction with each peripheral is modeled in C on the PC

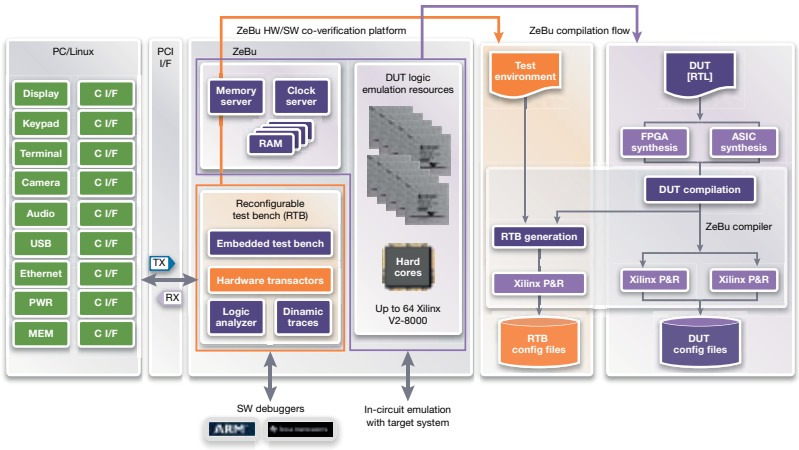


Figure 2: The back end that converts high-level commands into bit-level protocols is mapped to hardware within the emulation system's RTB architecture

Emulation transactors (VIP) typically are comprised of three elements. At the core is a synthesizable protocol specific element, usually a BFM or full IP implementation that is placed in the emulation hardware along with the DUT. Advanced systems like ZeBu have dedicated resources for these elements, to optimize the performance in the system. (Figure 2) In the normal two-way data flow, communications between the host and DUT in the emulator are transaction-based, maximizing the system performance. On the downstream side, the protocol block converts the transaction-level signals to pin-level signals, and interfaces to the DUT's protocol specific physical interface.

A well architected emulation system can accommodate dozens of such protocol specific transaction-level interfaces.

The beauty of transaction-based verification methodology is that all the interfacing from the DUT to the external test environment is software configurable and downloadable. Changing the system from one block to another, or testing multiple blocks in parallel, or even shifting from one SoC design to another, can all be done through software configurations from anywhere on a network. The system, accessed as a networked resource, offers much more flexibility and value than if used for ICE based verification.

Use the Right Methodology at the Right Time

A factor in obtaining the best out of the emulation system is to use a mix of verification methods appropriate to different stages on a project. During early architectural design, high-level models in electronic system level tools (ESL) are used to make tradeoffs and optimize different parts of the design. With much of today's SoCs consisting of major blocks being re-used from prior designs or licensed from third parties, there is considerable RTL available very early in the project.

In such cases, a hybrid ESL—emulation environment can be used where the RTL models can be exercised in the emulator, and the blocks of the design are exercised in the ESL tools. The full visibility into the RTL within the emulator can prove extremely useful in identifying implementation problems in the RTL blocks while exploring your design alternatives.

Once RTL is all available, typically block-level designs are initially tested with simulation. Once the bug discovery frequency drops to a reasonable level, maybe a bug a day, users frequently will move the block-level testing to an emulation system and begin running more exhaustive tests at speeds unattainable with simulation. At this point, firmware may be introduced to verify the initial hardware—software interactions.

After running initial regression tests in simulation on the entire SoC, many teams quickly move their full SoC testing to emulation where they can greatly expand the real-time cycles on their designs. Typically at this point, early versions of drivers and other low level software are available and testing can begin moving into realistic system test scenarios.

When the RTL design is stable enough, it's time to give emulation system access to the software development teams, whom up to that point may have been using non-cycle accurate ESL models for development. It may also be optimal to provide multiple, high performance FPGA-based prototypes, like our HAPS systems, to the software team to accelerate their development.

Conclusion

Getting the most value and productivity of your emulation system generally requires that you

- Leverage virtual test environments to simplify the use model and increase accessibility
- Adopt the most appropriate verification methodology at the right time to optimize your entire verification flow.

Innovation… Is Good Verification!

X-Propagation is innovative technology—it takes a traditional verification approach that is time-consuming, overly pessimistic, and late-stage, and changes 'x' verification into a fast-running, easy-to-deploy, early-to-adopt part of the verification process. As chip complexity continues to rise, we must become increasingly adept at disrupting traditional verification methods and deploying unique solutions that shift the paradigm.

Static, formal, hardware-accelerated, software-optimized, pre-compiled solutions and more will need to be explored to address emerging problems. Deep integration among tools and flows will also need to be addressed to increase the efficiency of these cross-platform, multi-disciplinary solutions.

At Synopsys our goal is seamless integration among verification platforms and disruptive technologies that address complex flows. Working closely with our customers we are solving not just today's problems, but those of the next five years.

Rebecca Lipon
Product Marketing Manager for Functional Verification

X-Propagation: Providing RTL Simulation-Based Resolution of 'x' Related Issues

Gate simulations are an onerous task that most verification teams still find necessary prior to the tape out of the chip. However, many of the design risks mitigated by gate simulations can now be addressed using RTL lint tools, static timing analysis tools, and logic equivalence checking. While these tools work well for validating RTL synthesis and final timing verification for example, the potential for optimism in the 'x' semantics of RTL simulation remains an issue that must be resolved. Most teams validate 'x' propagation in gate-level simulation, but gate simulations are time consuming, tedious to debug, and overly pessimistic with respect to 'x' on re-convergent paths, which can result in simulation failures that do not represent real bugs. Finally, gate simulations can only be performed later in the simulation cycle since one needs a gate-level netlist, meaning that this time-consuming methodology for resolving x-propagation issues often delays the critical path to tape out. VCS now provides a new add-on technology, X-Propagation, which attempts to model 'x' behavior more accurately at the RTL. X-Propagation can be used to reduce and potentially eliminate gate-level simulations for 'x' validation.

Why do X's occur in RTL?

Before describing more about the technology, I want to review the four main reasons why a logic variable may have the value 'x':

1. **Model:** a model may drive 'x' when its behavior is not known or an error condition has occurred. This could be the result of protocol or timing violations.
2. **Explicit RTL Assignment to 'x':** designers may assign the outputs of their circuit to 'x' as a means of expressing an "output don't care" condition. Logic synthesis tools use the freedom of "output don't care" conditions to minimize the logic.
3. **Testbench:** the bus protocol may specify that a given signal should not be consumed under some conditions (e.g. valid=0). The testbench can drive 'x' into the DUT to ensure that it is indeed not sensitive to the signal value.
4. **Uninitialized state:** all flip-flops and memories in a design start with the value 'x' until they are initialized through a reset or a write of a non-X value

RTL constructs can be ambiguous with respect to 'x'. A few key RTL constructs which have optimistic 'x' propagation semantics are:

1. `If/else` statements
2. `case` statements
3. Bit Selects and Indexing
4. Ambiguous edge transitions

```
reg state;   // Note : state is not reset
#define INIT_STATE 0
#define RUN_STATE 1
always @( posedge clock )
  if ( ( state == `INIT_STATE ) && ( !start_input ))
    state <= `INIT_STATE;
  else if ( ( state == `INIT_STATE ) && ( start_input ))
    state <= `RUN_STATE
  else if ( ( state == `RUN_STATE ) && ( !stop_input ))
    state_<= `RUN_STATE;
  else if ( ( state == `RUN_STATE ) && ( stop_input ))
    state <= `INIT_STATE;
  else
    state <= `INIT_STATE
```

Figure 1: If/else 'x' optimism in RTL code

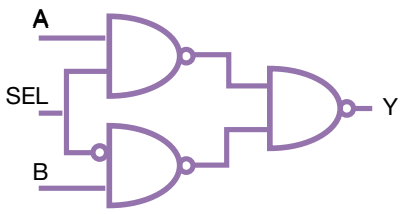


Figure 2: Simple combinatorial logic block

A	B	SEL	Y		
			Gate Sim	X-Prop: T-Merge	X-Prop: X-Merge
0	0	X	0	0	X
0	1	X	X	X	X
1	0	X	X	X	X
1	1	X	X	1	X

Table 1: Comparing T and X-Merge results with Gate Simulation Semantics for simple logic block

I will not walk through all of these in detail in this article, but I will detail one coding example with respect to if/else statements in Figure 1 to help highlight how often these semantics may occur in RTL design. Take a simple state-machine with two states, and suppose the designer accidentally forgot to reset the state variable:

If `start_input` and `stop_input` are initially zero, then on the first clock edge of RTL simulation the `state` flop will be 'x' and the predicates to all of the if statements will also evaluate to 'x'. As a result the final `else` statement will execute and cause `state` to get initialized to `INIT_STATE`. The resulting circuit, however, will not force any initial condition on `state` and 50% of the time the circuit will power up in the `RUN_STATE`, which could result in a dead-lock if the assertion of the `stop_input` depends on the state starting in `INIT_STATE`. It is actually rather easy to create simple circuits that could easily result in different behavior with respect to 'x' propagation in RTL versus actual circuits. Because of this, a robust methodology must be deployed to help catch 'x' optimism and ensure proper functionality.

How Does VCS X-Propagation Mitigate RTL 'x' Risk?

The VCS X-Propagation Add-On changes the way 'x' is simulated to remove the optimistic 'x' behavior modeled by standard Verilog semantics. When simulating with X-Propagation, if the conditional predicate of an `if` statement evaluated to 'x', it will propagate to the variables that are assigned in both the `if` and the `else` branches. Similarly, if a case expression evaluates to 'x', the 'x' propagates to variables assigned in the case statement. Ambiguous edges on clocks are handled by considering the behavior when there are only definite edges (e.g. 0->1) and the behavior when there are ambiguous edges (e.g. X->1) and merging the results. The key to X-Propagation semantics is an ability to merge multiple values which could be assigned to an output variable.

VCS X-Propagation supports two different merging algorithms: the T and X-Merge options. When an if statement is evaluated using X-Propagation semantics, the values of the assigned outputs are initially calculated both on the if and else branches. When X-Propagation is configured to use T-Merge semantics, if both potential output values are the same, then the simulator accepts this known value as the final output. With the more pessimistic X-Merge semantics, whenever the conditional expression evaluates to 'x', the assigned outputs become 'x', even if both branches would resolve to the same assigned value. X-Merge semantics are closer to what would be observed in a gate-level simulation.

A robust simulation methodology is necessary to address the risk associated with 'x' optimism in classic RTL simulation. Gate simulation, the traditional approach to address this problem, has three major drawbacks :

- It starts late in the verification cycle since it require a gate net-list
- Gate simulation times are long and tedious to debug
- There is intrinsic 'x' pessimism on re-convergent paths which can lead to false failures

RTL simulation using VCS' X-Propagation semantics addresses these issues. First, X-Propagation simulations can be performed early in the verification cycle. Any modeling issues related to the testbench can be quickly identified as soon as the team brings up the simulation environment. This methodology enables 'x' debug activities to occur on RTL rather than gates, resulting in reduced time to debug. Generally, a VCS X-Propagation simulation with T-merge semantics is less pessimistic than gate simulations, eliminating false failure. It is important to note that X-Propagation simulation semantics are focused on identifying a specific class of problems related to X-semantics in RTL; therefore it is not necessary to run all RTL simulations using this technology. Ideally this technology should be deployed when RTL is brought up, and again when RTL is approaching the first netlist drop.

One Final Note

The VCS X-Propagation technology works with all flows including coverage and debug. Support for VHDL will be added in the VCS 2013.06 release. This technology has been deployed on production-level designs since 2010. Contact your local support team if you would like to try the VCS X-Propagation Add-On.

Testing, Testing...

The tasks surrounding PCI Express verification fall into two main buckets: Verifying a digital core, which will require compliance tests and verifying the integration of a previously verified core within a larger system, which calls for integration testing. Each category of tests differs in scope and each present its own unique challenges. With the growing levels of adoption of reusable IP from highly credible commercial providers, such as Synopsys, most engineers will face integration testing.

The problem comes when the integration testing re-uses a methodology and scope designed for compliance testing. This often results in unnecessary amounts of testbench redesign, over-verification and debug.

The solution is simple, yet often overlooked: selecting the right VIP with features targeted at the proper category of testing. The proper combination of verification strategy and IP can help achieve superior product quality, as well as a much more efficient flow.

Neill Mullinger
Product Marketing Manager for Verification IP

Leveraging Advanced Verification IP Capabilities to Accelerate PCIe Integration Test

The complexity of most external protocols and the ready-availability of thoroughly-proven commercial IP means that most systems companies now use pre-built IP cores and PHYs rather than build in-house. This choice has significant implications for verification teams since a core from a trusted source, or one being reused from a previous design does not require full compliance testing; that should have been completed as part of the core's progress to certification. However, these cores do need integration testing, which differs from block-level compliance testing. While integration is simpler it is certainly not trivial; verification teams still need to run extensive real-life traffic, cover common error cases, and apply relevant application transfers.

Similar to the way design teams benefit from design IP, the verification team can greatly benefit from Verification IP (VIP). Protocol-based VIP can have a big effect on schedule by providing the features, tests and debug capabilities to make the process run smoothly.

A major challenge for verification teams is where to draw the line when verifying integration: there is a vast gap between the two extremes of a few simple connectivity tests and full re-compliance checking of the core. This article gives a few pointers on what should be included in integration test and how to get it done efficiently.

Determine Which Tests to Run?

For the purposes of integration, compliance should be assumed, so the end goal is to move sufficient traffic (TLPs) from point A to point B to validate configuration, connectivity and system integration. Verification IP can have a huge effect on how simply this can be accomplished and there are a few key features worth mentioning that greatly affect the task of integration test. In particular, the ability to auto-generate traffic, inject errors, check the protocol, debug issues and meet coverage goals. This five-point test strategy achieves the goal of protocol verification.

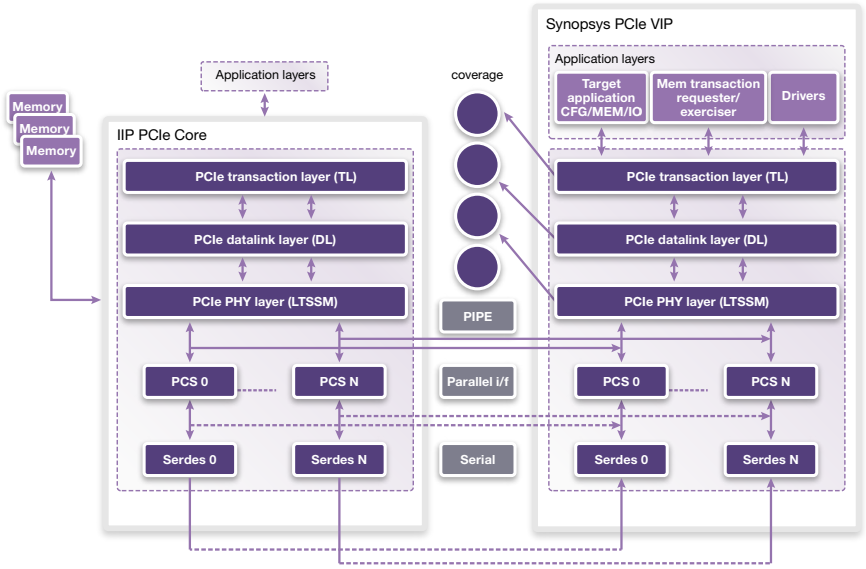


Figure 1: VIP hook-up at PIPE, parallel or serial i/f

Basic Connectivity/Link Training and Initialization

The first sets of tests are basic connectivity and link training to make sure the system is 'wired' together and can train the link. Typical tests will include:

- Verify Supported Speeds: Bring the core and VIP out of reset and have them train the link to the desired speed and number of lanes. Once the LTSSM reaches L0, re-negotiate the link with the various supported speeds
- Lane Reversal: Assuming support for lane reversal, reverse the device wiring and bring up the link to verify lane reversal is properly supported.

General Traffic Testing

The fundamental goal of integration testing is the ability to send TLPs end-to-end. The following sets of tests will verify that this capability is sound

- VIP as requester
  - A series of config writes and reads (to the same address). The built in VIP scoreboard will use a shadow memory to validate the correct response and flag mismatches.
  - A series of writes/read and io/writes reads (if supported) to the DUT.
  - Alter the above tests to vary between minimum and maximum data payload.
  - Setup for a series of writes and reads. This will mimic real traffic—both foreground and background to exercise the DUT. Any violations will be flagged by the VIP.
- Core as requester
  - Respond to the valid address range for the EP/RC, also the VIP should be set for the min/max completion size in bytes and max payload.
  - Do a series of DUT writes and reads such that the VIP as a completer responds. Completions will vary as per settings to verify the DUT can handle multiple completions with varying payload size.

Interface Testing

The following set of tests will verify the different interfaces available within PCIe sub-system.

- Tests should be repeated for all supported interfaces: PIPE, Parallel, and Serial.
- Lane Error Handling: Error injection should be done on specific lanes such that the link renegotiates to a lower number of lanes. i.e. if two lanes and the 2nd lane has an EI, the link should renegotiate to a single lane. A large variety of error injections should be done on the PL and other layers.
- VC to TC mapping: VIP should be configured to match the core in terms of the VC to TC mapping. Once setup, traffic should be injected and the queue usage verified.

What to Look for in the Verification IP

The VIP should have all the built-in capabilities to achieve traffic generation, error injections and handling, checking, coverage and debug with the minimum of effort by the user. The Synopsys PCIe VIP has been architected to provide all of the features needed to simplify and accelerate integration test.

Traffic Generation

The Synopsys PCIe VIP provides includes a series of software applications that serve as an application layer above the TL. Applications enable transaction based verification rather than forcing users to build their own TLPs. One is able to use the Driver to drive all the PCIe transaction types, the Requester to generate a series of reads/writes to memory in the background, the Target Completer for automation of completions, and the NVMe application for handling of connected SSDs. These applications go far beyond the generation of TLPs, as they provide automated scoreboarding and the ability to fine tune error injections.

Error Injection

With the Synopsys PCIe VIP there is no need for complex callbacks to handle every aspect of injection, checking, and recovery; built in error injections handle this automatically. Users take advantage of predefined error injections—set up to occur randomly. The VIP will inject, detect, and attempt to recover.

A good example would be the injection of an LCRC error. The VIP verifies the DUT

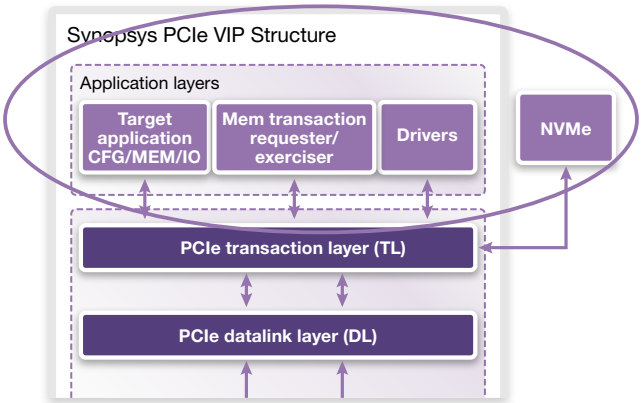


Figure 2: Higher level applications

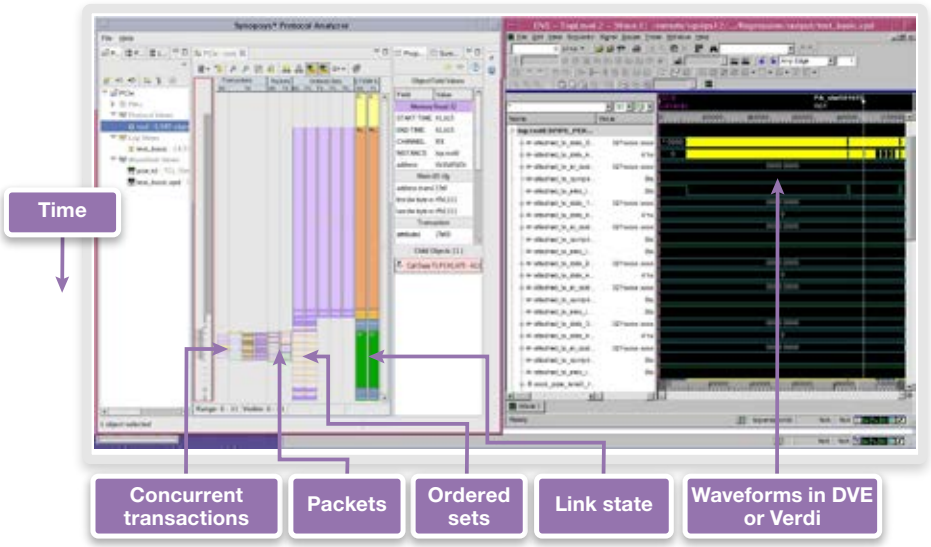


Figure 3: Protocol debug made easy with Protocol Analyzer

responds appropriately (NAK). The model will then attempt to retry the transaction. All of this is done with minimal coding, typically one or two lines of code. Should the DUT not NAK the transaction, the model will flag an ERROR.

Automated Self-Checking

It is important for the VIP to be on the look-out for inappropriate / unexpected behavior from the DUT. This may be that the DUT unexpectedly transitions from L0 to Recovery or an unexpected NAK. Many things may go wrong and it is critical that the VIP is always observing and flagging errant conditions.

Functional Coverage and Statistics

The PCIe VIP verifies that all key-behaviors of training, traffic flow, and error injection have taken place. It provides two forms of functional coverage: one being SystemVerilog functional coverage groups and the other generic protocol statistics. Both give a unique viewpoint of training, data flow, and error conditions. Covergroups track across the TL, DL, and PL, as well as the PIPE interface. Coverage will show TC to VC mappings, details on the TLPs and DLLPs, completion statuses, etc. Statistics provide a different perspective not so well suited to the binning defined in SV covergroups. Statistics provide counts of the error injections, TLP types and higher-layer application activity. Together they give a comprehensive view of what happened over the course of testing.

Analysis Capabilities

The Synopsys VIP contains features that ease analysis of activity and faults. First, there is the standard log output that provides multiple levels of verbosity for tracing Link State, TLP/DLLP Data Flow, and Error conditions. Then there is the transaction log which mimics the output of a bus logic analyzer.

Finally there is the Synopsys Protocol Analyzer which provides a unique, protocol aware view of the PCIe.

Summary

Integration test of a PCIe core has a different scope than compliance testing, however it does not make the task by any means trivial. The Synopsys PCI Express VIP includes the features needed to quickly accomplish the task in hand.

Debug: Changing Radically

Debug productivity has always been a significant, if somewhat hidden, issue in chip verification. Roughly 35-40% of verification resources (and time) are devoted to debug- a figure that has remained steady over the last decade.

However, as the leading edge of design has moved from ASIC to SoC, key tasks such as problem Identification and root cause tracing are growing in complexity. Factors such as power, protocol compliance, software implications, and testbench issues are all playing significant roles in making debug more complex and expensive.

The Verdi[3] debug platform, now a part of the Synopsys family of verification products, iss designed to help you tackle these issues head-on. This adjoining article is the first of many to come that will relay helpful practices in the use of Verdi[3]—we look forward to helping you 'Live long and Debug!"

Thomas Li
Product Marketing Director for Debug

An Open Debug Platform: How to Improve Debug Productivity with Verdi Interoperability Apps

SoC design teams rely heavily on design knowledge to develop an understanding of the intent of the designer and the causes of design behavior. Design engineers need to understand the design structure and intended behavior in order to complete their portions of the SoC design and successfully integrate other components. Verification engineers must understand the design intent and critical aspects of the design structure in order to craft effective verification environments, checkers, and tests. For the engineers responsible for SoC debug, an understanding of both familiar and unfamiliar parts of the design (and their behavior) is essential to tracking down the root causes of unexpected behaviors and implementing changes so designs behave as intended.

Harnessing Design Knowledge

Design knowledge can also be used to assess whether the design complies with specifications or project requirements and to accurately transform data for input to downstream design, verification, or analysis steps during the SoC flow. The range of potential design knowledge applications is literally unlimited and commonly includes:

- Traversal of design structure, including modular hierarchies and gate-level netlists
- Traversal of design behavior by examining signal values over simulated time
- Correlation of verification results to design structure
- Correlation of component groups by user-defined criteria

Of course, it is possible to gain design knowledge by manually opening and examining the various design and verification files described earlier. However, this approach is extremely laborious and impractical for all but the simplest designs. Indeed, by today's measures, modern designs of even moderate complexity mandate the use of automated programs and utilities to expedite viewing, tracing and analysis of design and verification data. Rapid, accurate development of such programs requires engineers with deep, rich experience and a knowledge-based infrastructure that:

- Automatically performs much of the necessary analysis of raw design and verification data
- Stores and preserves knowledge, not just data, including correlations between elements
- Accesses knowledge via application programming interfaces (APIs) at the right granularity
- Allows intuitive use of standard viewing tools, such as source code, waveforms, schematics, and state diagrams

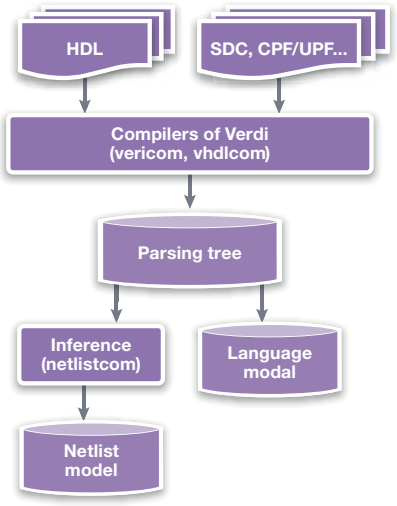


Figure 1: VIA data models

Verdi³ Automated Debug Platform is a highly automated debug system that accelerates design comprehension of complex IP components, design modules and entire SoC designs. Built upon a unified design database (KDB), Verdi compiles, extracts and preserves the design, simulation and analysis data needed to reveal the functional operation and interaction between design, assertion and system testbench elements.

Leveraging the Database

The Verdi Interoperability Apps (VIA) platform is an interface that enables both end users and application developers alike to leverage the power of the KDB/FSDB for data mining and manipulation based on their specific design/verification tool and SoC flow requirements.

KDB contains design structure information from the compilation of design sources in HDL stored in a compact binary data format. The design data is displayed by the Verdi software in multiple design views and used for a variety advanced debug functions, such as hierarchy tree display, source code analysis and automatic schematic generation, etc.

Through the VIA platform, users have access to two types of data models as shown in Figure 1. Language models allow users to query the design information at the source code level. For example, it allows users to traverse the design hierarchy, search particular instance by name…etc. Netlist models hold the design information as extracted by the design inference engine. This model contains four distinct types of objects that will be extracted from the design (Figure 2).

- Instance
- Port
- Instance Port
- Net

To the right is an example on how to list all the registers in the design after inference:

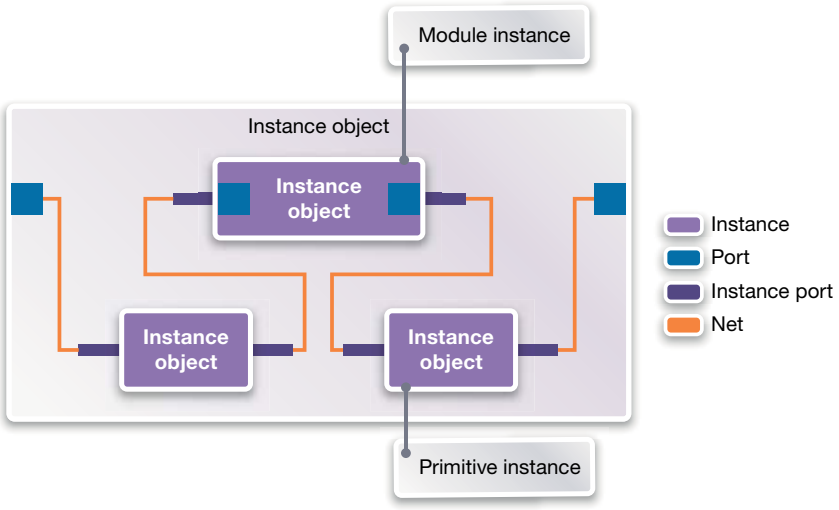


Figure 2: Data objects in netlist model

```
proc get_registers { } {
set file [open "file.log" "w"]
set itr [npi_nl_iterate -type npiNlInst -refHandle ""]
set instance_scan [npi_nl_scan -iterator $itr]
while {$instance_scan != ""} {
  set name [npi_nl_get_str -property npiNlCellType -object $instance_scan]
  if {$name == "npiNlModuleCell"} {
     traverse_sub $instance_scan "file"
  }
  if { $name == "npiNlFlipFlopCell" || $name == "npiNlLatchCell"} {
     puts $file [npi_nl_get_str -property "npiNlFullName" -object $instance_scan]
  }
  set instance_scan [npi_nl_scan -iterator $itr]
 }
 close $file
}
proc traverse_sub {instance_scan file_name} {
  upvar $file_name aaaa
  set itr_sub [npi_nl_iterate -type npiNlInst -refHandle $instance_scan]
  set sub_instance_scan [npi_nl_scan -iterator $itr_sub]
  while {$sub_instance_scan != ""} {
  set name [npi_nl_get_str -property npiNlCellType -object $sub_instance_scan]
   if {$name == "npiNlModuleCell"} {
      traverse_sub $sub_instance_scan "aaaa"
   }
   if { $name == "npiNlFlipFlopCell" || $name == "npiNlLatchCell" } {
     puts $aaaa [npi_nl_get_str -property "npiNlFullName" -object $sub_instance_scan]
   }
  set sub_instance_scan [npi_nl_scan -iterator $itr_sub]
  }
}
```

Figure 3: VIA application case study—automatic X value debugging

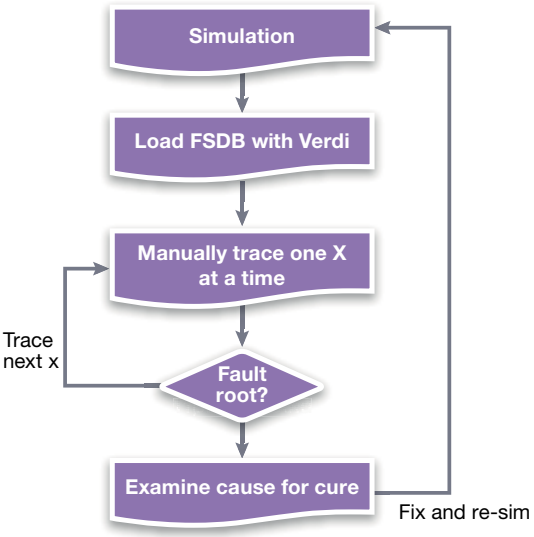


Figure 4: Traditional X value debug flow

```
01 sidCmdLineBehaviorAnalysis
02 sidCmdLineBehaviorAnalysisOpt -bdb_load work.lib++/work.bdb -incr -clockSkew 0
-loopUnroll 0 -bboxEmptyModule 0  -cellModel 0 -bboxIgnoreProtected 0

03 debImport "-top" "system"
04 debLoadSimResult ./wave/waveX.fsdb

05 tfgTrX -noBBox -snapVC -causeCnt 1 -batchInput .xlist.txt  -batchOutput tracex.rpt

06 debExit
```

Figure 5: Automatiac XTracing example code

Figure 4 shows the typical debug for tracing the root cause for X (unknown) values observed from the simulation results today.

Usually, a designer can list all the X values within a specific design scope or simulation time period when bringing up the post-simulation waveform. Then the X values must be traced individually to isolate the root cause candidate. If the found candidate is not the true root cause, then the process is repeated for the next X path, until the real source is found. It is pretty normal that designers will see hundreds of X values appear in their simulation results. It may take, therefore, a large number of iterations to identify the root cause—a very time consuming process if performed manually. It will be ideal if we can automate the process- to have a program that can read in all the X values from the simulation result and perform the following steps automatically:

1. Read in design and simulation results from KDB and FSDB
2. Generate the list for all X found from the simulation
3. Automatically trace all the X values to their sources
4. Generate a report for the tracing result

The example code in Figure 5 above showed that the automation can easily be done via a small Tcl script. VIA Tcl interface provides direct access to all Verdi features, like Behavior Analysis (e.g. sidCmdLineBehaviorAnalysis in line 01, or traceX e.g. tfgTrX in line 05).

Summary

SoC design and verification requires analysis of vast amounts of correlated data about the structural composition and temporal behavior of designs. The complexity of this challenge requires a variety of commercial tools and custom utilities that work together reliably in user flows.

Verdi's VIA platform furthers the EDA industry's paradigm shift toward greater openness and interoperability to help SoC development teams address the unique requirements of their design and verification flows. By providing access to the design knowledge platform of the industry's most popular debug software, the VIA platform enables design and verification engineers to rapidly create custom applications that are optimized to save time and resources and easily deployed for a more automated, interoperable SoC flow.

Upcoming Events 2013

Verification Seminar Series
Various locations WW
February-December
(Check with your local representative)

HW/SW Verification Symposia
Various locations WW
March–August
(Check with your local representative)

MIPI Alliance World Congress
Barcelona, Spain
February 25-28

PCI-SIG Developers Conference
Tel Aviv, Israel
March 11-12

SNUG Silicon Valley
Santa Clara, CA
March 25-27

SATA Plug Fest
Taipei, Taiwan
April 23-26

Design Automation Conference
Austin, TX
June 2-6

PCI-SIG Developers Conference
Santa Clara, CA
June 25-26

Intel Developers Forum
San Francisco, CA
September 10

SNUG Boston
Boston, MA
September 12

SNUG Austin
Austin, TX
September 18

ARM TechCon
Santa Clara, CA
October 29-31

Resources

Functional Verification
www.synopsys.com/Tools/Verification/FunctionalVerification

Debug
www.synopsys.com/Tools/Verification/Debug

Verification IP
www.synopsys.com/Tools/Verification/FunctionalVerification/VerificationIP

Hardware-Based Verification
www.synopsys.com/Tools/Verification/Hardware-verification

Synopsys SolvNet
solvnet.synopsys.com

Share this by email

Feedback and Submissions

We welcome your comments and suggestions. Also, tell us if you are interested in contributing to a future article. Please send your email to **avb@synopsys.com**.

SYNOPSYS®
Accelerating Innovation

Synopsys, Inc. • 700 East Middlefield Road • Mountain View, CA 94043 • www.synopsys.com

02/13.TT.CS2661/CPR/500.