Hello and welcome back to Verification Horizons. Over the past few years, I’ve come to really enjoy this particular issue because it gives me a chance to reflect on everything that happened at DAC. Since DAC seems always to be the focal point of our industry’s year, I find it fascinating to see how things have progressed from year to year—within the industry at large and especially here at Mentor Graphics. The central focus of Mentor’s verification activities at DAC this year was our annual luncheon, in which we discussed the topic of “Innovations in Verification.”

Aside from giving me a chance to present to my largest audience of the year (this year, we had a standing-room-only crowd of over 300 attendees!), it’s a wonderful opportunity for me to immerse myself in the full spectrum of Mentor’s verification offerings. I then have the chance through this newsletter to share some of the same thoughts with our ever-expanding (over 35,000 and counting) online audience. So, this quarter’s issue will continue with the same theme.

Perhaps the greatest verification innovation to occur in the past year, if I may say so myself, is the Open Verification Methodology (OVM), which we released in partnership with Cadence back in January. The success of this partnership continues with the recent release of OVM 2.0, which strengthens many of the key features of OVM. Our first article, “A Quick Tour through OVM 2.0,” provides an overview of some of the new features of this exciting release.

Hello and Welcome Back to Verification Horizons!

By Tom Fitzpatrick, Editor and Verification Technologist

“From its inception, we have always envisioned OVM as a “methodology platform,” in that it provides the infrastructure necessary to incorporate additional technologies in a unified way to extend existing verification strategies.”

—Tom Fitzpatrick
From its inception, we have always envisioned OVM as a “methodology platform,” in that it provides the infrastructure necessary to incorporate additional technologies in a unified way to extend existing verification strategies. To that end, our next article discusses how we’ve seamlessly integrated the inFact® intelligent testbench automation tool into OVM environments. The modularity of OVM allows inFact to plug into existing environments so that it looks like just another stimulus generator to the rest of the testbench. However, the integration allows users to take advantage of inFact’s superior stimulus generation capabilities, which we’ve discussed in several past issues as well.

Our next technology article focuses on the ability of our 0-In® CDC verification solution to provide silicon-accurate RTL simulation of designs with asynchronous clock domains. By combining both static and dynamic clock-domain crossing analysis, we see how metastability issues can be accurately modeled, even in pre-synthesis RTL simulation. If you have multiple clock domains in your design—and who doesn’t these days?—you’ll want to take a look.

As more and more capability gets pulled into the Questa® verification platform, it becomes critical to provide an easy-to-use graphical environment to analyze and debug the results of your verification runs. Our next article highlights many of the new debug and visibility features of our Questa 6.4 release, with particular emphasis on the class-based debug capabilities. In addition to supporting general SystemVerilog object-oriented testbenches, we’ve enhanced Questa’s ability to support OVM testbenches by letting you see OVM-specific elements in intuitive ways.

Our next two articles address innovations in productivity for SoC verification. First, we take a look at how Questa Codelink™ improves the overall verification flow by letting you run and debug real software in your RTL environment. The key innovation here is the ability to fully trace all aspects of the processor’s behavior and present it in a familiar, software-centric debugging environment, right alongside the rest of the design and testbench, and to play it back interactively to speed debug. Our Seamless® product also allows the processor model to communicate with other parts of the testbench, at the transaction level in OVM, enabling performance profiling and other advanced analysis.

We follow this with an in-depth look at our Veloce® SoC emulator, which combines compilation and runtime performance with a simulation-like debug and analysis environment to maximize productivity by minimizing turnaround time. Veloce’s TBX interface lets you connect the emulator to your existing OVM environment at the transaction level, minimizing testbench overhead and letting your design run at emulation speeds while you still have access to the full breadth of options in your OVM testbench. See how it all fits together?

The success of OVM is further chronicled in our next article, written by our friends at LSI in Boulder, CO. It shows how LSI was able to bridge the knowledge gap between the verification team, who wrote the OVM environment(s), and the designers, who had to be able to write tests without getting bogged down in the details of the environment itself. I think this article does a great job of highlighting the advantages inherent in OVM’s architecture of separating stimulus sequences from the testbench hierarchy. Sequences (as with scenarios in OVM 1.1) are relatively straightforward to write and, as LSI shows, can be packaged and reused hierarchically to build up complex tests from simpler ones. The designers have to focus just on the stimulus they want to generate and simply plug those sequences into the environment built for them by the verification team. This is exactly how we envisioned this feature to be used, so it’s great to see it being put into practice.

Our “Partners’ Corner” section includes four articles from some of our Questa Vanguard Partners program. First, our friends at Doulos continue their “A Practical Guide to OVM” series with a discussion of assembling the verification components, running a test, and then reconfiguring the verification environment. Next, Agnisys presents a “Closed-Loop Requirement Verification” methodology, which they developed as consultants to automatically keep the verification plan in sync with evolving requirements throughout the verification process. After this, Synterix Technology shows how to use OVM to address the conflicting goals of the verification team and the RTL designer. Their approach is similar to LSI’s, which just goes to show the merits of this approach. Last but not least, our friends at PDTi discuss their SpectaReg tool, which auto-generates components and tests to support the verification of addressable registers in a design. The web application automatically creates OVM-compatible (and other
flavors of) deliverables for all aspects of the register flow. It’s always great to receive so many high-quality articles from our partners, and I encourage you to submit your own.

We conclude this issue with our “Consultants’ Corner” article, in which our Mentor Consulting group shares with you some of the success they’ve had deploying Mentor Multi-view Verification Components (MVCs) with our customers. MVCs are yet another example of the OVM methodology platform, providing the means by which we can make more advanced tools and techniques seamlessly fit into your verification flow.

Putting each issue of Verification Horizons together is truly a team effort, but that team is led by my Program Manager, Rebecca Granquist. Rebecca recently went on maternity leave just as we were putting this issue to bed, which gives the rest of us a whole new appreciation for all she does to make things run so smoothly around here. Please join me in wishing all the best to Rebecca and her growing family.

But, just like the New England Patriots, who won yesterday after losing MVP quarterback Tom Brady for the season, there’s no substitute for a good team, where everyone knows what to do and can “step up” to fill in when necessary. That’s kind of like what we’re trying to do here at Mentor, too. We want to give you everything you need to be successful, with the right architecture and methodology so you can plug in whatever tool or technology you need to get your job done. I hope this issue of Verification Horizons has shown this to be the case.

Respectfully submitted,

Tom Fitzpatrick

Verification Technologist
TABLE OF CONTENTS

Page 7...
A Quick Tour Through OVM 2.0
by Tom Fitzpatrick, Verification Technologist, Mentor Graphics

Page 10...
Using inFact in an OVM Environment
by Mike Andrews, Testbench Automation Applications Engineer, Mentor Graphics

Page 13...
Does Your RTL Simulation Accurately Predict the Behavior of Your Silicon
by Rindert Schutten, Sr. Marketing Manager, Mentor Graphics

Page 16...
Understanding and Debugging the Verification Environment
by Stephen Bailey, Production Marketing Manager, Mentor Graphics

Page 18...
Processor Driven Verification – Use it for More Than Just Sign-off
by Piotr Luszczak, Technical Marketing Engineer, System-Level Engineering Division, Mentor Graphics

Page 24...
The Innovation Behind Veloce: Time for a Closer Look
by Vijay Chobisa, Technical Marketing Manager; Charley Selvidge, R & D Director, Mentor Graphics
Bringing Everyone on Board with OVM
by Julie Ferski Blair, Staff IC Verification Engineer, LSI and Dan Cohen, Application Specialist, Mentor Graphics

Partner’s Corner:

A Practical Guide to OVM – Part 2
by John Aynsley, CTO, Doulos

Closed Loop Requirement Verification
by Anupam Bakshi, Agnisys Technology

Concurrent Verification: Resolving Conflicting Goals
by Bruce Bergenfeld, Chief Technology Officer, Synterix Technology

Automated Register Abstraction Coming to an OVM Near You
by Jeremy Ralph, President, Productivity Design Tools Inc.

Consultant’s Corner:

Off to the Races with MVCs
by Mark Peryer, Verification Practice, Mentor Consulting
“At Mentor, we want to give you everything you need to be successful, with the right architecture and methodology so you can plug in whatever tool or technology you need to get your job done.”
A Quick Tour Through OVM 2.0
by Tom Fitzpatrick, Verification Technologist, Mentor Graphics

INTRODUCTION
Since the initial release back in January of the Open Verification Methodology (OVM), development and collaboration between Mentor Graphics and Cadence has continued its brisk pace. Through some great interaction with OVM users, and guided by the recently-formed OVM Advisory Group (OAG), we’ve focused our recent efforts on the successful release of OVM 2.0, in which you’ll find many usability enhancements, bug fixes and other general improvements. In addition to improving the code, OVM 2.0 also includes the anxiously-awaited OVM User Guide, which provides step-by-step instructions and guidelines for applying OVM most effectively in the development of verification components, testbench environments and general verification IP.

In addition to the User Guide, OVM 2.0 improves the implementation of OVM sequences by adding TLM interfaces and other ease-of-use features from OVM scenarios. This “unified sequences” implementation preserves the best of both sequential stimulus mechanisms while simplifying the use-model and making sequence/driver interaction a conceptual superset of the existing TLM communication facility. We have also added explicit support in the OVM factory for the creation and overriding of parameterized classes and added much-requested debug methods to trace port/export connectivity throughout the testbench hierarchy.

THE OVM USER GUIDE
The OVM User Guide is a valuable tool for both new and advanced OVM users. Organized in a straightforward manner, the guide presents all the important OVM concepts in an easy-to-follow format that takes you through everything you need to get going with OVM. The guide contains the following sections:
• OVM Overview—introduces the OVM concepts and how they all fit together
• Transaction-Level Modeling—explains the concepts and mechanics of how TLM interfaces work and how it can be used to improve the modularity and reusability of verification components
• Developing Reusable Verification Components—discusses the essential building blocks of reusable verification components (sequencers/sequences, drivers and monitors), shows how to encapsulate them into agents, and shows how to incorporate multiple agents into reusable environments.
• Assembling Testbenches and Writing Tests—discusses the steps required to specify the components, environments and stimulus sequences necessary to make your tests do useful work with a minimum of effort.
• Advanced Topics—a detailed discussion of some of the more advanced concepts in OVM, like component phasing, the OVM factory and sequence layering.
• Xbus Example—the well-known xbus example updated to use the new OVM2.0 features.

UNIFIED SEQUENCES
The major feature of the OVM 2.0 release is the use of TLM interfaces to implement the sequencer-driver communication semantics. The ovm_seq_item_prod/cons_if classes in OVM 1.1 have been replaced with the ovm_seq_item_pull_port/export, which extend the familiar TLM put/get/connect concepts to the realm of sequences. We have also added the ability to specify arbitration priorities for multiple sequences, which was previously only available for ovm_scenarios, and preserved the familiar sequence registration and selection mechanisms.

With the addition of TLM interfaces, sequencers and drivers may now be parameterized, so the communication between them is automatically type-specific. A driver has a built-in ovm_seq_item_pull_port, called seq_item_port and a sequencer has a built-in ovm_seq_item_pull_export that implements the necessary communication methods. These interfaces implement a superset of the familiar TLM calls, so a TLM-based driver can interact directly with a sequencer.
class simple_driver extends ovm_driver # (simple_item);
  simple_item s_item;
  ...
  task run();
    forever begin
      seq_item_port.peek (s_item); // notice: no cast
drive_bus(s_item);
      seq_item_port.get (s_item); // s_item ignored
    end
  endtask

The peek() call provides the item to be processed via the drive_bus() call (or any user-defined functionality). After the item is processed, the get() call removes the item but, since it was already processed, it may be safely ignored.

For users of sequences in OVM 1.1, the same sequencer/driver interaction calls are also implemented:

class simple_driver extends ovm_driver # (simple_item);
  simple_item s_item;
  ...
  task run();
    forever begin
      seq_item_port.get_next_item (s_item); // notice: no cast
drive_bus(s_item);
      seq_item_port.item_done();
    end
  endtask

In addition to the request semantics described here, we've also added a built-in response mechanism to be used when the driver must send a transaction back to the sequence. The sequence_item now includes both a sequence_id and a transaction_id field, which is automatically created in the sequencer for the request but must be set by the driver for the response. When the driver sets the response, sequence_id field to match the request.sequence_id, the sequencer is able to route the response back to the appropriate sequencer. Depending on the application, driver may put the response transaction back to the sequencer in one of several ways:

    seq_item_port.put(rsp);
    rsp_port.write(rsp); // via analysis port
    seq_item_port.item_done(rsp);

Which path to use is up to you. The seq_item_port is bidirectional and supports both put() and item_done(). The rsp_port analysis_port is built into the ovm_driver and is available if desired.

As with other TLM connections, the seq_item_pull_port/export connection is done in the parent's (component or env) connect() method:

class simple_env extends ovm_env;
  virtual function void connect();
    driver.seq_item_port.connect(sequencer.seq_item_export);
  endfunction
endclass

Using TLM interfaces to implement the sequence-specific semantics as well as standard put/get/peek methods allows sequencers to be connected to drivers that may choose to use only a subset of sequencer/driver methods. So, an existing driver that uses get and put to manage requests and responses can be ported to use sequences in OVM 2.0 by simply replacing the existing TLM interface(s) with the ovm_seq_item_pull_port.

Along with adding TLM functionality to the sequence/driver interaction, we restructured the sequence/sequencer API as well. The 'ovm_do*' macros are still available, but since they have been simplified, it is now much easier to write a sequence without them. The steps required for a sequence to send a sequence item to a sequencer are as follows:

1. Create the req item using the factory.
2. Call wait_for_grant().
3. Optionally randomize req.
4. Call send_request(req).
5. Call get_response(rsp).

Depending on the application, you are free to add your own functionality at different points in this algorithm. These steps are also encapsulated in the ovm_do* macros, which provide user-definable callback hooks as follows:

• pre_do() called between steps 2 and 3
• mid_do() called between steps 3 and 4
• post_do() called between steps 4 and 5

We've also added an explicit start() method to sequences that allows them to be started explicitly on any sequencer directly.

virtual task body();
  fork
    seq_a.start(seqr1);
    seq_b.start(seqr2);
  join
endtask
The `start()` method takes a sequencer as an argument and is a blocking call. Multiple sequences may be started in parallel using fork/join*. When a sequence is supplied as an argument, the `ovm_do*` macros will start the sequence on its p_sequencer. There are also a new set of `ovm_do_on*` macros that are similar to the existing `ovm_do*` macros, but take an additional sequencer argument.

**PARAMETERIZED CLASSES IN THE FACTORY**

Because every specialization of a parameterized class in SystemVerilog is actually a unique type, it is necessary in OVM 2.0 for the factory to create and override types and instances on a type-specific basis—not just via a string-based name lookup. In addition to allowing the factory to return an instance of the desired type directly (avoiding the need to call `$cast`), this new mechanism avoids the possibility of hard-to-detect bugs that could crop up due to typos in strings. With the addition of parameterization to sequencers, sequences and drivers, this capability is even more important.

Using the type-based factory, components are created using a static allocation function. Instead of the string-based

```verilog
$cast(driver, create_component("my_driver","driver"));
```

the component (or object) is created directly

```verilog
driver = my_driver::type_id::create("driver");
```

The `type_id` is a “wrapper” class that is automatically declared when the component or object is registered with the factory. The `create()` method is a static method that returns an allocated pointer to the desired component/object.

Type-specific factory registration can be done using the `ovm_*_utils` macros (which also automatically create the `type_id` wrapper). Questa does not require the macros, but to ensure your code will run on both Questa and Cadence’s IUS, you should use the macros for now. For registering parameterized types, a specialized set of `ovm_*_param_*_utils` macros has been created (the wildcard can be “component,” “object,” or “sequencer”) and must be used in place of the `ovm_*_utils` macros for parameterized classes.

Type-specific overrides may be set for all objects of a specific type

```verilog
factory.set_type_override_by_type(orig_type, override_type, bit replace = 1);
factory.set_type_override_by_type(driver #(d_type)::get_type(),
new_driver #(d_type)::get_type());
```

or on a per-instance basis

```verilog
factory.set_inst_override_by_type(orig_type, override_type,
string inst_path);
factory.set_inst_override_by_type(driver #(d_type)::get_type(),
new_driver #(d_type)::get_type(),
"d1");
```

In each case, `get_type()` is a static method of the type (created by the macro or declared manually), and “factory” is a pointer to a singleton `ovm_factory` instance, which manages all of overrides. More information about this new functionality may be found in the OVM User Guide.

**CONCLUSION**

From its inception, OVM has been a collaborative effort, not just between Mentor and Cadence, but across the industry. Through robust interaction with our users, on-site, through the OVMWorld Forum (http://www.ovmworld.org) and now also including the OVM Advisory Group, the OVM is uniquely positioned to continue to grow and adapt to the evolving needs of our users. The progress made from the initial release less than a year ago, through OVM 1.1 and now OVM 2.0 is further evidence of the ongoing success of that vision. We are extremely pleased with these new additions to the OVM and hope you will be too.
Using inFact in an OVM Environment

by Mike Andrews, Testbench Automation Applications Engineer, Mentor Graphics

INTRODUCTION

The Open Verification Methodology (OVM) provides a framework that includes both a methodology and code libraries that a verification engineer can use to build a testbench that is modular, interoperable, and reusable. It includes mechanisms for creating the various components and other objects that are required and for communicating information between these components.

The inFact® intelligent testbench automation tool allows the user to build testbench components whose activity is controlled by one or more rule graphs that define the verification scenarios and stimuli that are to be applied to the device under verification (DUV) or to one of its interfaces. At simulation run-time these rule graphs interact with the tool’s intelligent algorithms and possibly other testbench components to efficiently achieve the required functional coverage.

The compiled binary rule graphs, known as testengines, are defined by describing all the legal sequences of activity that form each verification scenario, with each step in the sequence corresponding to an action (in inFact terminology). Each action in turn is linked to code in the form of a task or function in the high-level verification language (HVL) code. The description syntax is an extended BNF style and, therefore, provides a very compact representation of both sequential operations and parallel choices, as well as other higher-level constructs, such as loops.

Figure 1

InFact supports the creation of these testbench components in many forms and in most HVLs, one of which is an OVM-style SystemVerilog class-based component; that is, a class that is derived from the ovm_threaded_component base class.

In an OVM testbench, the stimulus is described using a combination of OVM sequences or scenarios and SystemVerilog constraints in the form of randomize-able variables and other data types. The success of the random stimulus generation is measured by using SystemVerilog’s coverage measurement capabilities in the form of coverpoints / covergroups and the cross-coverage of these. By taking advantage of the modularity and standard communication mechanisms that are characteristic of the OVM, the stimulus generating components in any OVM testbench can easily be replaced by inFact graph-based components. The graph-based components provide dramatic efficiency improvements in reaching coverage, with their benefit scaling logarithmically as complexity increases, and they are often simpler to code than their constrained random based counterparts.

BUILDING THE inFACT OVM COMPONENT

As mentioned previously, as is the case for most OVM testbench components, the inFact stimulus component is implemented as a SystemVerilog class that extends from the base class ovm_threaded_component. The inFact component creation integrated development environment (IDE) allows the user to select this style from a pull-down
The inFact OVM component generally contains one or more master ports of, for example, the `ovm_blocking_master_port` type, or any other OVM TLM port type. It may also contain slave ports through which it can be controlled or provided information from a higher-level component, such as a Test Controller block (which can also work under graph control).

To make the integration into an OVM environment easier, the inFact component creation IDE builds a template for the class definition when the Project Build menu option is executed. Templates for the action functions are included within the file. The class definition is defined inside a package for easy inclusion in the testbench environment.

The architecture of the testbench, the number of DUT interfaces that are to be driven, or the number of TLM ports associated with those interfaces, determine the appropriate number and type of TLM ports that need to be instantiated. This requirement is no different from any OVM testbench component that drives transactions at the transaction level.

**Figure 3**

list of the supported options, and a template for the class definition is generated by the IDE from various user inputs. The user fills out this template to produce the complete component class definition.

The inFact component’s functionality is split between the class itself and an associated interface. At a basic level the class contains the definition of the testbench component and the tasks that are mapped to the actions in its rule graph(s), and the interface instantiates the testengine(s).

Note that, while it was previously mentioned that the testbench stimulus components are typically at the transaction level, the lower level drivers could also be constructed so that they too can be driven under the control of a rule-graph, allowing them to operate autonomously to produce all the possible transaction types they support. This allows the testbench to potentially be directed to obtain verification coverage in that context: i.e., to ensure that the interface to which the driver is connected operates correctly across the full range of the transactions that it is intended to support.

**Figure 4**

**COMMUNICATING WITH OTHER TESTBENCH MODULES**

Any communication between the inFact verification component and other elements in the testbench follows standard OVM mechanisms.

The basic unit of communication in an OVM-style testbench is some form of transaction data structure that may be passed from component to component, either by copying it and ‘put’ing it onto a TLM channel via the appropriate port, or by ‘put’ing a reference to it (a pointer). This is no different when using inFact, and, in a similar manner as that employed in a constrained-random methodology, the testbench stimulus generation component will first build the data
structure by making choices for the transaction parameters and then send the transaction (i.e., ‘put’ it) through the port onto the channel for consumption by another component.

In the inFact graph-based methodology, this is done by using combinations of the previously mentioned actions and special actions called meta_actions. The actions map to atomic testbench activity, which in this context could be a choice between a read or a write transaction, with the associated code setting the ‘READ’ or ‘WRITE’ attribute of the data structure. Meta-actions allow the definition of a range of integers, with the range being context-specific according to where it is defined in the possible sequences (or paths) through the graph. If, for example, the possible addresses for a read are different from those for which a write is valid, then meta_actions, such as setReadAddr and setWriteAddr, can be created with the appropriate ranges. The tasks associated with these meta_actions will then get a value from this range passed into them by the inFact algorithms, and the address attribute of the transaction data structure can be assigned this value.

Figure 5

Once the data structure is constructed, the transaction is sent by a subsequent action, called send in this example.

Figure 6

An important difference between inFact’s algorithms and a typical constraint solver for constrained-random testbenches is that inFact always creates unique tests, even when there are multiple parameters that make up the transaction or data packet definition. This means that, from a coverage perspective, each cross product of any number of variables will be generated once and only once, unless the user directs the tool to do otherwise. In a typical application this can provide savings in the order of 95 to 99 percent in the time it takes to reach coverage goals. This could mean that multi-week verification runs could be reduced to a few hours, allowing teams to expand the scope of their verification significantly, therefore reducing defect density.

SUMMARY

Using the inFact tool as the methodology to create verification scenarios and stimulus is an excellent and easy addition to an OVM-based environment. The inFact tool set generates OVM-style component templates, which can be easily connected to an existing testbench to quickly achieve dramatic increases in coverage closure.
For decades designers have relied on software simulators to predict silicon behavior, so few question whether the software models used to represent silicon behavior are sufficiently accurate to correctly predict silicon behavior. In fact, today’s verification methodology is based on this foundation — simulate the logic to ensure the right function is being computed, then use static timing to ensure the logic performs its function within the timing constraints.

So why question whether simulation accurately predicts silicon behavior? The fundamental reason is that the way engineers architect their designs today differs from the designs they did 10, or even 5, years ago. Clearly, designs are much larger and more complex, but that in itself does not cast doubt on whether software simulation is effective at predicting silicon behavior. The reason to revisit this question is that some of the fundamental assumptions that allow the “simulation + static timing” methodology to be effective no longer hold with today’s design styles.

**THE HEART OF THE PROBLEM**

The validity of the RTL-based verification methodology rests on the fact that the simulator is allowed to assume (because this assumption is checked by static timing analysis) that at each clock edge the data computed by the combinational logic is stable in time; thus, it can be “clocked” into the receiving registers.

However, for many of today’s high-performance, low-power designs, a simulator simply is not allowed to make this assumption, at least not for all registers. The reason is that these designs employ several asynchronous clock domains and that the registers, that connect these clock domains, will occasionally be clocked while the data on their inputs has NOT yet stabilized. When this happens, the simulator will compute a value that may differ from the actual behavior of this register in silicon. What actually happens in silicon is that the outputs of these registers become metastable. This means that both the timing and value are non-deterministic: the register’s value can settle to either a logic ‘0’ or a ‘1’. Hence, the simulation results, which are deterministic, do not always match the actual logic value the silicon produces. As a consequence, any RTL model of a design that contains these clock-domain-crossing (CDC) registers is incomplete in capturing the behavior of physical hardware, and, as such, is incapable of accurately predicting silicon behavior.

### CONFINING METASTABLE VALUES

Signal values that neither represent a logical ‘0’ or a ‘1’ are obviously undesirable in any digital design. Designers, aware of the phenomenon of metastability in CDC registers (of course), have devised special design structures, called synchronizers, which are typically used whenever data is transferred between two clock domains. The purpose of these synchronizers is to confine the extent of metastable signals to the internals of the synchronizer. In this way, synchronizers prevent metastable signals from contaminating the rest of the design with ill-defined logical values.

Synchronizers, clearly an essential aspect of multi-clock designs, however are only part of the solution to address metastability related issues in the design. The non-determinism introduced by metastability is fundamental, so while synchronizers do a good job at confining the extent of metastable signals, they do have non-deterministic propagation delays. When comparing the silicon behavior of a synchronizer versus the results of an RTL simulation of that synchronizer, the simulation results can still be +1 or -1 cycle off (in terms of the destination domain clock). Hence, even for designs that use synchronizers, RTL simulation is NOT an accurate predictor for silicon behavior.

### A COMPREHENSIVE SOLUTION

The Mentor 0-In® CDC verification solution, together with an RTL simulator, provides all the capabilities needed to deliver silicon-accurate RTL simulation of designs that have asynchronous clock domains. The 0-In CDC solution automates three verification steps:

1. Verifying that proper synchronizers are present between asynchronous clock domains — this is a static analysis step
2. Verifying that synchronizer-specific data transfer protocols are implemented correctly — this is a combination of static and simulation-based analysis
3. Verifying the overall functionality of the design using metastability-aware, silicon-accurate simulation — this is a simulation-based verification step

---

**Does Your RTL Simulation Accurately Predict the Behavior of Your Silicon?** by Rindert Schutten, Sr. Marketing Manager, Mentor Graphics
STATIC CDC ANALYSIS

Simply by reading in the RTL representation of the design, 0-In CDC automatically identifies all clock domains and subsequently all registers that connect the asynchronous clock domains. It also verifies that these registers are part of well defined synchronizer structures. Through a dedicated GUI-based debugging environment that automatically classifies all CDC registers and synchronizers designers can easily debug and resolve incorrect synchronization between clock domains.

The next step in the analysis process is to ensure that the synchronizer-specific data transfer protocols are implemented correctly by the design. A synchronizer-specific data transfer protocol is a set of rules that precisely specifies the behavior the design must adhere to, both in the source and destination clock domain, to ensure that data is transferred correctly, irrespective of whether metastability occurs, across that synchronizer. Take for example a relatively simple situation where data is transferred from domain A to domain B where the clock frequency in A is higher than the clock frequency in B. Clearly, to ensure no data loss, the input to any register connecting A and B must be held stable long enough (in terms of clock cycles in A) so that this register can be certain to capture this value when it is triggered by the clock in B. How long this value must be held stable depends on the clock frequencies in A and B, and is exactly defined by the rules comprising the data transfer protocol.

As the result of the comprehensive static analysis, the set of protocol assertions capturing the rules the design must follow to ensure that no data loss occurs when transferring data from one clock domain to the other are atomically generated by 0-In CDC.

CDC PROTOCOL VERIFICATION

Executing a test on the design with the protocol assertions included provides valuable information to the designer. First, these assertions will pinpoint potential design errors because they are automatically triggered when the transfer protocol is not adhered to by the design. Second, the coverage metrics on these assertions measure the completeness of the tests (with respect to data transfer protocol verification).

METASTABILITY-AWARE SIMULATION

During the analysis phase all CDC registers are identified. This is exactly the set of registers that require metastability-aware modeling to ensure silicon-accurate RTL simulation. In the 0-In CDC verification...
solution metastability-aware modeling of the CDC registers is accomplished by including \textit{behavioral metastability models} (BMM) associated with the CDC registers in a simulation.

A BMM is completely complementary to the RTL code and consists of two parts: a passive assertion that monitors whether metastability conditions are present and a driver that directly interacts with the simulator kernel to impose the $+1$ or $-1$ cycle delay on the output of the register. Then, when during simulation, the assertion detects that metastability conditions are present, it will automatically trigger this driver imposing randomly an adjustment of the delay through the register.

Running an RTL simulation with the BMMs included will identify the functional issues the metastability can cause and 0-In CDC offers extensive GUI-based analysis and debugging tools complementary to the simulation debugging environment to help pinpoint the root caused of those problems.

Moreover, just as for the protocol assertions, the extensive coverage metrics on the assertions in the BMM form a quality metric on the completeness of the tests CDC registers. By keeping track of whether and when the delay through a CDC register is modified (because metastability conditions are present) the coverage results will immediately expose the completeness of the tests in their ability to expose bugs in the design that originate from non-deterministic delays through CDC registers, and prompt the designer to either extend and enhance the test so that all corner cases are covered.

**CONCLUSION**

Today’s high-performance, low-power designs employ many asynchronous clock domains. Straightforward RTL simulation of these designs cannot accurately predict silicon behavior, opening the door for critical bugs to slip through the verification process.

To prevent this from happening, designers need to deploy a solution that accurately models the impact of metastability on the behavior of the design. To this end, they need to model the registers that connect asynchronous clock domains, in a silicon-accurate fashion.

The 0-In CDC verification solution provides all the capabilities needed to selectively deliver metastability-aware, silicon-accurate simulation. In fact, its extensive static analysis and built-in coverage metrics make it the only comprehensive solution to fully address the challenges of today’s multi-clock designs.

---

**Figure 2.** Today’s RTL verification methodology (simulation + static timing) is very effective for single clock domain designs but falls short in accurately predicting the behavior when asynchronous clocks are involved. \textit{Metastability-aware simulation, by explicitly modeling the non-deterministic behavior of CDC registers through BMMs, is the only way to achieve silicon accuracy in an RTL simulation environment.}
Understanding and Debugging the Verification Environment
by Stephen Bailey, Product Marketing Manager, Mentor Graphics

As more and more users deploy object-oriented design techniques to develop reusable dynamic verification components at the transaction-level, the more important it becomes to provide debug and analysis capabilities that work at that level of abstraction and verification environments composed of class-based objects. You must be able to debug at multiple levels of abstraction, using the same models at the transaction, register transfer, and gate levels. Transaction-level modeling (TLM), abetted by standards-based verification components, delivers the required capacity, visibility, and performance to thoroughly debug today’s extremely complex designs.

These complex systems also impact the nature of verification by requiring powerful and equally complex verification environments. This complexity makes it increasingly difficult for any single person on the verification team to grasp the entire structure of the verification environment. To enable a meaningful understanding of the verification environment, your simulator should recognize verification components derived from base classes and display them as part of the overall simulation hierarchy, as shown in figure 1.

Visualization of the verification environment within the hierarchy of the overall simulation provides a familiar place to orient the structure of both the testbench and the design and allows you to use that orientation to easily navigate to the corresponding source and the objects, processes, and threads local to a testbench or design scope. A hierarchical view of the verification environment also shows relationships between verification components.

To facilitate reuse as well as debugging of classes, it is important to understand the inheritance structure of the classes that are available for reuse and that are in use within the testbench. Figure 2 shows the graph-based view of a class hierarchy.
Another aspect of class-based verification environments is the heavy use of threads instead of static HDL processes. Technically all threads are dynamic, even if they are used in a static manner. Threads can suspend for multiple reasons and have a subprogram called stack associated with them. Debugging the verification requires access to information on the current state of all threads in the simulation, the ability to view the call stack of the threads (the thread-local variables), and the ability to control which thread will get executed next. Figure 3 depicts a thread/process window where all of these capabilities are provided.

In order to debug simulation results, it is necessary to view relevant aspects of the verification environment; such as configuration information, stimulus generated, and design responses received. If the testbench is operating at the transaction level, then viewing results at an RTL signal/clock accurate level of detail requires the user to mentally bridge the gap from the transaction to the register transfer level.

As transactions are transacted over multiple clock cycles and often involve related transactions (child or initiator/response relationships) that would also be transacted over multiple clocks, manually identifying transactions and their relationships to signal activities can significantly hinder debug productivity. To facilitate transaction-level debug, your simulator must support transaction recording and viewing for both SystemVerilog and SystemC.

Other features to look for in a simulator include a waveform window, delta and event viewing, and relationship recording. When logged along with related signals, transactions viewed in a waveform window facilitate debugging the testbench and design at both the transaction and register transfer levels of abstraction. Delta and event viewing allow you to understand the ordering relationships between untimed events. Relationship recording with transactions makes it easy to highlight the relevant cause and effect relationships in transaction streams. When used with Verification IP that capture standard bus protocols, bus protocol stacks are recognized and the correlation between transactions and RTL signal activity is explicitly highlighted, greatly simplifying debugging bus protocol problems. Figure 4 shows transaction and signal viewing in a wave window with the protocol stack highlighted for a sequence of AXI transactions.

These advanced debug capabilities combined with high simulator capacity and speed are essential to successful verification of today's complex, mixed-language designs. In order to achieve the most efficient and effective debugging and verification, you need an environment for design and testbench debugging that enables you to work at a higher level of abstraction while providing automation to improve productivity and advanced capabilities that allow analysis and quick identification of bugs.
Processor Driven Verification – Use it for More Than Just Sign-off
by Piotr Luszczak, Technical Marketing Engineer, System-Level Engineering Division, Mentor Graphics

Current techniques of applying test vectors from an HDL testbench only begin to mimic processor bus behavior. The introduction of processor driven testbenches into the existing verification methodology enables real-world verification and extensive re-use of testbench software throughout the project. One of the limitations to effective utilization of processor driven tests has been the difficulty debugging software in a processor running in a logic simulator. This paper presents proven methods of debugging and trace technology which overcome the limitations to enable the benefits of processor driven tests throughout various hardware and software integration stages of your project.

WHAT IS PROCESSOR DRIVEN VERIFICATION?

Today’s complex circuit designs with embedded processors require execution of instructions in software to meet the maximum coverage of SoC. There are several classic approaches to achieve these goals. Driving bus cycles into the hardware design, execution on a full functional model, compiled drivers to target, and testbenches in C or assembly code are among those methods. Figure 1 illustrates the structure of a traditional verification environment for testing an image processor circuit.

However, this approach has drawbacks as it is incomplete. The tests do not cover entire interaction between the processor and external logic. And there is no software execution as part of the entire system. Since the end product is processor driven, this "real simulation" approach to verification is superior to traditional bus-functional models.

An example of processor driven test bench is shown on Figure 2. The test is stored in program memory. The processor executes the test, updates image processor registers, and sends noisy picture into a source frame buffer. After processing the image, the output is stored in the destination frame buffer. The processor test software then examines the enhanced picture.

PROCESSOR DRIVEN TEST SETUP

Questa Codelink offers an effective, non-intrusive, processor driven verification environment. The setup process requires few, if any, changes to the traditional RTL flow. The HDL instantiation of the processor may be replaced by a DSM (Design Simulation Model) which is a compiled, encrypted RTL sign-off model. These models are provided by core vendors and represent the golden (RTL) behavior of the processor. The test as a C program is compiled using a processor’s native compiler to an image file containing symbol information (ELF). Figure 3 shows an example of the configuration flow.

Figure 1. Classic test bench of SoC design

Figure 2. Processor driven testbench
DIFFERENT TYPES OF CODE HAVE DIFFERENT COVERAGE PROFILES

The effectiveness of this verification process, and therefore the ability to find flaws before tapeout, is strongly influenced by the type of code being executed. Directed tests offer block-level and system-level testing. They enable specific module or function verification as well. Diagnostic code exhaustively tests hardware interfaces and some functionality. For more extensive functional verification, driver code enables extensive functionality coverage and some interface coverage.

ADVANTAGE OF PROCESSOR DRIVEN TESTS

The current methodologies and flows require simulating the entire SoC design with the sign-off processor model prior to tapeout. Since these are trusted models from the foundry or processor vendor, the proven accuracy is assured. Thus, the processor model and the test code perform a role of a highly leveraged and portable test bench. Figure 4 shows the range of usability of processor driven tests throughout an SoC project.

Questa Codelink offers comprehensive debug visibility as another benefit in addition to traditional approaches to SoC verification techniques. Most of the relevant pins and registers are accessible by looking into an HDL waveform file. The instruction execution trace file is difficult to interpret by manual inspection (Figure 4).

Debugging a full functional processor model when it goes into “flat-line” becomes a challenging task. The trace file includes a list of fatal errors while the waveform shows no processor pins activity for a number of clock cycles. Relying on these two simulation outputs exclusively can be very cumbersome and tedious.

A simple, conceptual example can demonstrate this. Let’s find a value of variable “n” in the following function:

```c
void set_reg (int a, int b, int c, int d, int n, int f)
{
    unsigned long value;
    for (f=0; f<4; f++)
    {
        value *= (unsigned long *) (0x80000000+(*f));
        value = (*unsigned long *) 0x60000000 = value;
    }
    *n = value;
}
```
First, we need to find the address where the function is active – PC register must be in this range, which we find in the symbol table:

Between L1.24 and L1.72 (labels assigned by the compiler) “n” is in R5. Outside of this range it is in memory at the stack pointer + 12 as we can see from the assembly code view:

The next step is to find the values for L1.24 and L1.72 in the symbol table listing:

Then we need to go to the waveform trace file and find a write cycle to register R5:

This gives some idea of the difficulty involved.

Questa Codelink simplifies all of these steps into one: hold your cursor over the “n” in the source view (Figure 6).

The debugging productivity of SoC Verification has been increased by adding software source viewer to the Questa functional verification environment.

Both C and assembly level are supported. Visibility into processor registers, memory and the stack is provided and they update dynamically. The waveform viewer cursor is synchronized with the software source window thus the wave cursor tracks your progress as you step through code. Likewise, dragging the waveform cursor will scroll the source view to the corresponding source line containing the instruction being executed (Figure 7).
Figure 7. Questa Codelink source level debugger

Existing software debuggers lack comprehensive representation of X-states that are all too common in logic simulation. Questa Codelink allows X-states to appear in register and memory windows.

The speed of a DSM is usually a couple cycles per second. Hence, many simulations where the DSM model of processor is involved run hours and sometimes days. Waiting hours to observe the execution of a few instructions, so they can be debugged, slows down the verification process significantly. Questa Codelink provides a post-simulation debug to overcome these obstacles. During simulation time Questa Codelink logs changes in the processor’s general purpose registers. The logging process has less than a 1% impact on simulation time. This is more than made up in the time savings of post-simulation debug when the verification engineer has the ability to step forward and backwards through the test C or assembly in a blink of an eye. The mechanism is highly interactive and allows one to traverse a 15 hour of DSM simulation in 5 seconds. Setting breakpoints in the hardware and software sources is supported as well.

Other advantages include the ability to log and debug multiple processors simultaneously. A separate set of source, register, memory, variable and stack windows are displayed for each of the processors. As with single processor systems, all processor debug windows and the HDL waveform viewer are synchronized in multi-processor systems.

**QUESTA CODELINK PRO SPEEDS SIMULATION**

Questa Codelink Pro offers the same debug as Questa Codelink but includes a cycle-accurate model of the processor core, which executes software 10,000x faster than a design sign-off model (DSM). Overall simulation speed improves by 5x to 10x depending on the ratio of SW execution to logic simulation. No software modification is required.

The acceleration mechanism relies on a faster processor simulator model and removing all hardware simulation cycles used for accessing memory to fetch instructions or read and write data. The technology uses mirrored software memory models that are accessible both by processor models and the logic simulator. By optimizing memory access, the number of bus cycles as HDL simulation events is reduced to only I/O operations (Figure 8).

A minor drawback of the approach is the limited visibility of processor-memory traffic.

**SEAMLESS – MORE FLEXIBILITY, MORE SPEED**

To get even more control over a debugging visibility and simulation speed, coherent memory is employed. Seamless uses coherent memory server technology to offload hardware memory accesses interactively. The user controls the optimized address spaces and the user may elect to have software and hardware events enable or disable the optimized address spaces. Thus, the ratio between simulation speed and debug visibility is fully scalable. Figure 9 shows the Seamless HW-SW co-verification environment.
Figure 9. Seamless allows a dynamic switch between detailed hardware verification and high-speed software execution.

Figure 10 shows the difference in communication speed between hardware and coherent address spaces. Program memory, source and destination buffer are mapped as an optimized memory access so that communication between these memories and the cycle accurate processor model can be over 1000 times faster than using hardware access.

Figure 10. Seamless Coherent Memory Advantage

Coherent memory access bypasses the need to generate logic simulation transactions. The total number of logic simulation events is reduced and simulation time is reduced. The ratio of memory access to logic simulation cycles varies across phases of a processor-driven test (Figure 11).

Figure 11. Ratio of optimized to non-optimized accesses

Another technique to speed-up execution of processor drive tests is compilation and running the software on a workstation. This method is called Host Code Execution (HCE). The cycle accurate model of processor is replaced by bus functional model. Host Code Execution runs much faster than machine code executing on an instruction set simulator (Figure 12).

Figure 12. Host Code Execution (HCE) in Seamless

Portions of the software for an embedded system can be implemented in a high-level language and tested directly with the logic simulation, without machine code available. A workstation’s native compilers are used to develop embedded software.

The HCE application can access address spaces implicitly and explicitly. The HCE Implicit Access mode enables testing code automatically generate hardware read or write transactions located outside program’s code and data space. In HCE Explicit Access mode, read or write transactions are driven by explicitly placing data access API calls in HCE testing program.

Seamless allows communication between the processor, memories and I/O devices both at the RTL and TLM level. The transactions can be derived from OVM or SystemC.
CONCLUSION

The Questa Codelink, Questa Codelink Pro and Seamless products enable the integration of software driven stimulus engines with your verification environment. The instruction speed scalability from sign-off accurate processor models (DSM), instruction set simulator (ISS) to host code execution (HCE) allows running software and hardware design verification tests at multiple abstraction levels. Software source level debugging, TLM support, and the system profiler significantly leverage test development productivity, improving overall functional verification coverage.

Design performance metrics like memory and cache transactions, software execution, processor and memory subsystem power estimation, bus load and arbitration delay are supported in the Seamless system profiler (Figure 14). This feature allows spotting throughput bottlenecks in a graphical interpretation.

Figure 13. Mixed pin-level and transaction level.

Figure 14. Seamless system profiler helps determine optimal design parameters and archive best end-product performance.
While chip and verification complexity continues to grow, time to market pressure requires that chip verification be completed on time and on schedule. To help their customers with these issues, emulation vendors face challenges of delivering faster runtime performance, addressing capacity needs, supporting different stimulus sources/methodologies, and offering simulation like ease-of-use and debug.

Providing a level of flexibility where users can employ multiple stimulus methodologies in the same project, or on the same model using single emulator platform is an important capability. As verification moves to higher levels of abstraction, hardware-assisted verification tools are required to handle verification environments using these advanced methodologies.

The key factors that drive a user’s decision to work within an emulation environment include:

- Verification productivity
- Stimulus and monitoring flexibility
- Accurately modeling the final device (synchronous and asynchronous behavior)
- Space and power

Taking the above criteria into consideration, Mentor Graphics has created the highly successful Veloce emulator. One of the key attributes of Veloce is its custom and highly optimized SoC chip technology – which is a primary reason why potential users should consider Veloce rather than an emulator using a commercial FPGA or building a custom processor.

The Veloce SoC, while FPGA-like in its computation resources, has a different kind of network for interconnecting the computation resources that is optimized specifically for faster compile time. The constraint in designing programmable logic core is not to build the most optimized commercial FPGA, but to optimize the logic for emulation applications using a distinctive interconnect network. Veloce SoC supports a capability to independently compile the logic part of the design from the communication part; these are unique steps that use distinct, system-level resources for predictable compiles.

Further, Veloce SoC is a dense chip with a dedicated interconnect network that delivers higher capacity utilization and a smaller footprint. Another aspect of the footprint is system-level packaging. Veloce has a larger number of logic boards packed inside the physical chassis with switching boards to flexibly interconnect these logic boards. Clocks are system-level functionalities and come from their own dedicated resources that allow Veloce to support line break pointing, stopping the clocks, and triggering capability.

Other vendors have built a smaller system and tried to connect them together for bigger capacity, which not only takes significant real estate, but also exposed reliability issues due to hundreds of cables.

**VERIFICATION PRODUCTIVITY**

Compilation, runtime, and debug are the three factors that contribute to overall verification productivity. The compilation speed and runtime performance are important; however, a rich debug infrastructure to quickly identify the cause of the failure is key to the turnaround time. Veloce’s simulation-like interactive debug and advanced graphical user interface are designed to accelerate the ability to analyze failures.

**Compilation**

Fast and predictable compiles are essential for faster turn around. Veloce’s fast compile can support three to four turns-a-day for a 30MG design. Each Veloce SoC takes less than five minutes to compile, whereas a comparable commercial FPGA takes approximately two hours.

Compile predictability means that for repeated compilations of a design to accommodate small changes, all compiles are functionally identical and have similar performance and capacity requirements. Veloce uses a patented algorithm called timing resynthesis to build emulation databases that implement a zero-delay semantic model for a design that is free of any possibility of setup or hold time violations.
**Runtime**

Overall test execution consists of test setup, test run, and results collection. Length of run is contributed by how fast the emulator runs, how long it takes to load and setup the test on emulator (download design, initialize states, manipulate memory contents, etc.), and upload the test results. Some users have a methodology where they run very long tests while others run many small tests. So runtime clock speed and speed of auxiliary services are important.

Veloce SoC is designed to deliver high runtime clock speed by implementing dedicated built-in interconnect and state-of-art fabrication technology. Veloce has a built-in rapid access to all of the memory modeling resources, all the state elements for set/get and force operations, and efficient chip download.

**Debug**

One aspect of debug is how easily the user can use the debugging infrastructure. Another aspect is how long it takes to get the data to start the debug. In applications where one can control the clocks, the Veloce standard visibility model gives full visibility and can go as deep as the user wants by virtue of continuous uploads. This also addresses the ease-of-use issues around accessibility of the data.

Time to visibility is another aspect of debug productivity. If the user wants to access full visibility of arbitrary signals, it’s very challenging to do so in a resource efficient way. Veloce delivers a software technique called Software State Replay (SSR) that re-computes all data for a chip from a limited amount of per chip storage data. The SSR has a set of independent computations for each chip that can be distributed across many CPUs or cores. The on-demand visibility further accelerates time-to-visibility by running SSR on the user selected signals and time windows.

**STIMULUS AND MONITORING FLEXIBILITY**

One of the more difficult problems of high performance verification is establishing the testbench environment and test strategy which has the ability to accept a variety of high performance stimulus. Whether the stimulus is from hardware protocol generators or hardware connecting to a live application using speed adaptors or software environments via transactors, the ability to combine these stimulus sources into one unified verification platform is a challenge.

Veloce is built on a single platform architected to optimally support both hardware acceleration and In-Circuit Emulation (ICE). This enables the user to perform the complete verification on one platform from block-level, to chip-level, to system-level verification. Veloce has a flexible infrastructure for interfacing with target systems. Veloce can connect targets to the interfaces, can connect targets with multiple asynchronous clocks, can accept clocks coming in, and source the clocks to the targets. If targets are static, then Veloce has the ability to stop the clocks. The clock stopping ability gives access to certain kinds of powerful debug capabilities (for example, line break point and inspect memory contents).

Sometimes stimulus models need to be software for a variety of reasons; these models can be the same models used in the simulation environment. Mentor Graphics has a high-speed soft testbench interfacing technology called “TBX” with an abstract way of implicitly managing communication between a testbench and DUT to accelerate transaction-based environment on Veloce. TBX allows building a testbench environment that is equally applicable to acceleration using Veloce TBX and a software simulator that is SystemVerilog compliant. This fosters interoperability with software simulators and a common verification environment between simulation and emulation. TBX is based upon SCE-MI 2.0 standard (supports SCE-MI 1.0 for backward compatibility).

Figure 1 illustrates a verification environment designed with guidelines for acceleration that can seamlessly work in simulation and acceleration. It is not uncommon for users to use a combination of hardware stimulus for some interfaces and software stimulus for other interfaces of the same design in order to optimize or accelerate verification.

![Figure 1. The unified verification environment using Veloce TBX and Questa.](image-url)
Assertions are ways of compiling in checks to monitor the activities in the design. Veloce natively supports assertion-based verification in acceleration and ICE; users can write assertions using PSL, SVA, OVL, and QVL.

Veloce provides flexibility on the type of stimulus the user would like to use: hardware-based, soft testbench-based, or a combination of both. Veloce is the best-in-class transaction-based accelerator. It enables users to build their verification environments at a higher level of abstraction, develop the tests at the application-level, and communicate to the DUT at transaction-level using protocol-based Transactors. This allows users to do the debugging at the transaction-level instead of pin-level.

Finally, Veloce is designed to handle advanced verification methodologies where a user can run the verification environment in a simulator such as the Questa product from Mentor Graphics, and seamlessly migrate to Veloce TBX for acceleration. Figure 2 illustrates this type of typical verification environment.

As system-level integration increases on an individual chip, the adoption of multiple interfaces and standards on a single SoC becomes more customary. There is a significant requirement for verification teams to have off-the-shelf solutions that provide complex stimulus and analysis tools for a variety of applications, such as multimedia, wireless, networking, bus protocols, and embedded processors. Mentor Graphics has a full range of in-circuit speed adaptors for vertical market applications.

Figure 2. The advanced verification environment.

Figure 3. Veloce’s vertical market solutions.
ACCURATELY MODEL THE FINAL DEVICE

SoCs have a large number of asynchronous clocks requiring verification tools to provide capabilities to verify the chip using a live environment, where various interfaces are operating using independent asynchronous clocks. All emulation vendors can model a single clock domain design to accurately match the behavior of the final device.

However, in a custom processor-based emulator, all the domains are run simultaneously every time there is activity in any one domain, which makes the evaluations synchronized across all the domains, leaving some verification uncovered for asynchronous clock domain designs. It is not possible to catch race conditions, re-convergence, FIFO overrun, and glitches because of the synchronized evaluation of all the domains. There is no evaluation independence between two domains.

Veloce models both single-clock and multi-clock domain designs accurately in the emulator. It allows a distinct, independent cycle-based computation for each distinct clock domain, which is important for maintaining the fidelity to see cross domain effects.

SPACE AND POWER

For a large emulation model, space and power consumption/cooling are a significant cost to the users. Mentor Graphics’ chip technology, system-level design, and software technology enables the building of a high-capacity system in a smaller footprint – which consumes significantly less power. Veloce takes five times less power compared to a processor-based solution with a 4X smaller footprint, which in turn, allows a bigger capacity system in a smaller footprint.

Veloce supports up to 512 MG capacity in ¼ of the footprint compared to competitor 256 MG capacity machines.

CONCLUSION

Veloce is a high performance, high capacity platform with a very small footprint. Its unique architecture gives users flexibility to build verification environments using hardware stimulus, software stimulus, or a combination of the two. Veloce’s dedicated debug/visibility infrastructure and simulation-like debug make it an easy-to-use verification platform.
Bringing Everyone on Board with OVM
by Julie Ferski Blair, Staff IC Verification Engineer, LSI and Dan Cohen, Application Specialist, Mentor Graphics

The LSI Boulder verification team decided to verify their latest IP core with OVM and Questa. Although the verification team understood OVM well and could develop the primary constrained random tests, it was necessary to develop an interface that test writers outside the verification team could use to develop their own tests. The test writers are experts in the design they verify but do not know and do not have time to train on OVM or SystemVerilog.

This article describes an approach that preserves the flexibility and capability of the constrained random environment while allowing directed or semi-directed tests to be developed quickly and easily by engineers who are not experts in OVM or SystemVerilog.

From the start it was clear that a constrained random environment would be required to verify the highly complex LSI IP core. It was also important that others, outside the verification team, could develop tests. This was required as designers, system architects, and engineers working in the lab with the final silicon would need the ability to simulate with this environment.

Some of the test requirements could be captured as covergroups, but there was a clear need for directed and semi-directed tests to reach corner case cover points, demonstrate bugs, and prove fixes to the designers. These directed tests were also needed to bring up complex IP before starting random traffic.
If these reasons were not compelling enough, the team faced the specter of an earlier, frustrated attempt to move forward with a different proprietary verification language by another verification team within the company. There were only a few engineers (the verification team) capable of writing tests in that environment, as they were the only people who understood the language and aspect orientated coding. This caused the environment to be abandoned before it had successfully verified a single design.

Initially, stimulus generation focused on getting the most out of the constrained random environment. The aim was to allow a test to be developed rapidly with a minimum of typing. The environment initialized the DUT and started random traffic. Tests could override the scenarios that generated the traffic, or they could override the transactions that these scenarios generated to change the stimulus. This worked for the verification team but required knowledge of both the verification environment and the components that should be overridden. This expertise was critical because if a mistake was made, it would not always be obvious what had gone wrong.

Thus, a different approach was needed before others started using the verification environment. Ideally, this new approach would provide a familiar interface for directed test writers while maintaining the power of the constrained random environment.

Scenarios and sequences are a key to this new approach as they both allow automation and reuse. In this case, scenarios were used because the LSI verification environment was developed with OVM 1.1. Once OVM 2 is released, LSI will move to sequences.

The verification team designed the environment with a scenario driver and controller connected to each interface of the design. The team also added scenarios that implemented the basic initialization requirements for the DUT which are executed, based on the configuration, in the run task of the extendable test base class.

The basic configuration of the environment was handled by configuration objects, which were loaded into the configuration space using set_config in the test. These configuration objects were retrieved by components and scenarios in the environment using get_config. Fields that can be randomized in the configuration object were randomized, but constraints were used to ensure that only legal combinations of configuration settings were selected.

The test base class generated a default configuration object in the build method and used set_config to place it in the configuration space. The base test performs several tasks: it resets the DUT, starts the DUT clocks, and performs initialization within its start task, based on the configuration settings.

TEST DEVELOPMENT

To develop a test, which is an extension of the test base class, the test writer must first consider the environment:

For each interface:
- Is initialization required?
- Should randomization of initialization be disabled or constrained further than the defaults?
- Should clock frequencies or reset delays be randomized or constrained for desired behavior of the test?

For the environment:
- Can the number of drivers for each interface be randomized?
- In addition to the number of drivers, do specific driver IDs need to be chosen?

If any of the default values need to be changed in the configuration object, the configuration object is extended in the test class and the default is overridden in the factory before super.build is called in the test class.

Once the environment is configured the test flow can be implemented. An individual test is encapsulated in a file. This file contains a package with the classes that define the test inside it. The file contains the configuration object that will configure the environment for this test (if the default values are to be changed) and a scenario that controls the rest of the test flow. This scenario can call other lower-level scenarios. The run method of the test executes super.run to bring up the DUT based on the configuration settings and starts the scenario in the test.

If the scenario in a test proves to be useful in other higher-level tests, the scenario is moved from the test file to a file that includes all of the reusable scenarios. This forms a library of scenarios that can be called.

A register test is an example of a test that does not require initialization to run. This has a configuration object that disables initialization for all the interfaces and selects the minimum level of drivers in the system. It also contains a scenario that runs a directed test. This scenario runs through the registers, writing a random value to each register, and then reads it back, masking the bits that do not accept write operations or that can not be read back reliably.

A basic random traffic test has a configuration object that allows initialization to run on each interface and selects a random number of drivers for the interfaces where that is appropriate. It also has a scenario that starts random traffic on each of the interfaces and a run method that starts this scenario.

Once basic tests have been developed and each interface has a number of scenarios that can be used to drive stimulus, the test writer
can stay at a higher level of abstraction. This is accomplished by reusing existing scenarios with different configurations or by using combinations of existing scenarios.

All of the test packages are imported into another package file; this is the only file in the environment that changes when a new test is added. This package, containing all of the tests, is then imported into the top-level testbench module.

This approach has proved flexible enough to allow one of the designers on the project to eliminate his block-level testbench and verify the basic operations of the block using the chip-level OVM environment. Top to block reuse!

There is a little difficulty getting configuration objects from the scenarios, as they are not derived from an ovm_component. (Only classes derived from ovm_component can call get_config directly. This also affected the decision to move reset and clock generation into the environment from the testbench module, so it could use the configuration settings.) Fortunately, each scenario must be associated with a scenario controller before start is called. This produces a public handle (m_scenario_controller) that can be accessed from inside the body of the scenario to access the configuration object:

```verbatim
task body();
  ovm_object environment_config;
  environment_config = m_scenario_controller.get_config_object("environment_config");
  if (environment_config.initialize_pcie_interface == 1) begin
      ........
  end
```

The decision to move the execution of the DUT reset, clock starts, and DUT initialization from the verification environment into the run method of the test base class was made to simplify test writing as well. If those steps were left in the run task of the environment, there would be no way to start the run task of a test after all of those tasks completed—that is, without extending the environment class and overriding the run method for every test. By putting the execution of the tasks in the base test, only the base test was extended and the run method overridden to ensure that super.run was called first.

In summary, this approach is a compromise between the ideal environment for the verification team—where the test environment initializes each interface and starts random traffic—and the interface that the test writers are accustomed to. Each test requires more typing to develop the required stimulus, but the environment is more likely to be widely accepted.

It is still possible to use set_type_override to change components or transactions within the environment. The environment can be extended to make large changes to the way the DUT is stimulated. This allows an expert user to develop very powerful tests, but this level of expertise is not required to write the majority of the test cases.
INTRODUCTION

This article is the second of a series aimed at helping you get started with OVM in a simple, practical way. The emphasis will be on the steps you need to take to write working code.

In the previous article we explored how to hook up a class-based verification environment to a module-based DUT (design-under-test), and also looked at the structure of an OVM transaction and an OVM verification component. In this article we will look at assembling the verification components, running a test, and then reconfiguring the verification environment.

This article was written just as version 2.0 of the OVM class library was being released. The code you see in this article will run with both OVM-1.1 and OVM-2.0, and can be downloaded from the Doulos website at www.doulos.com/knowhow.

CONNECTING UP THE ENVIRONMENT

In the previous article we examined a driver component, which was one specific verification component instantiated within the verification environment. Now we will look at how the environment is constructed from this and other components. Again, we will describe the code line-by-line.
put export at one end through which transactions may be added to the FIFO, and a get export at the other through which transactions may be removed. In OVM, as in SystemC, a port can be bound to an export such that a method call made through the port is implemented behind the export.

```pascal
function new ( string name = "my_env", ovm_component parent = null );
super.new ( name, parent );
endfunction: new
```

If you read the previous article, this constructor should look familiar. It is important to always pass both the name and the parent arguments to the superclass constructor super.new.

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

m_stimulus = new ( "m_stimulus", this );
m_fifo = new ( "m_fifo", this );
endfunction: build
```

The build method of class my_env creates instances of classes ovm_random_stimulus and tlm_fifo by calling their constructors, remembering to pass the instance name and parent component, this, as constructor arguments. At this stage, the verification components have been created, but not connected together.

```pascal
$cast(m_driver, create_component( "my_driver", “m_driver” ));
endfunction: build
```

The build method also creates an instance of the my_driver component, but unlike ovm_random_stimulus and tlm_fifo above, not by calling the constructor. Instead, the driver is instantiated by calling the create_component method, which uses the factory to perform polymorphic object creation. In other words, the actual type of the object being created is set at run time such that the line of code shown above will not necessarily create a component of type my_driver. The factory makes it possible to override the type of component being created at run-time such that my_driver could be replaced with a modified driver for one or more specific tests. This overriding is actually done from within the test class, as we will see below. The factory is one of the most important mechanisms in OVM, permitting the creating of flexible and reusable verification components. This flexibility is not available when directly calling the constructor new, which always creates an object of a given type as determined at compile time. For example, m_fifo = new would always create an object of type tlm_fifo.

The arguments passed to the method create_component are the name of the component as registered using the ovm_component_utils macro and the local instance name of the component being instantiated.

```pascal
virtual function void connect;
m_stimulus.blocking_put_port.connect ( m_fifo.put_export );
m_driver.get_port.connect ( m_fifo.get_export );
endfunction: connect
```

The connect method is the callback hook for the second of the standard phases, and is used to connect ports to exports. The connect method is called after the build method, so you know that the components being connected will already have been instantiated, whether by new or by the factory. Notice that the method used to make the connections is also named connect. As well as for connecting ports to exports, the connect method may also be used to connect exports to exports going down the component hierarchy, and to connect ports to ports going up the component hierarchy.

```pascal
virtual function void start_of_simulation;
m_stimulus.set_report_id_action( "stimulus generation",
OVM_NO_ACTION );
endfunction: start_of_simulation
```

The start_of_simulation method gives an opportunity to set variables and execute code immediately before entering the simulation phase proper. In this example, the start_of_simulation method is used to set the action of the ovm_random_stimulus component by calling a method of the report handler. The method set_report_id_action modifies the actions to be executed whenever a report of a given type occurs within the given component, the report type being “stimulus generation” in this case. The flag OVM_NO_ACTION suppresses all actions, that is, silences the reporting of messages of this type from ovm_random_stimulus.
The run and report callback hooks are also included for completeness, although they do nothing but print out a message showing they have been called. Report is called near the end of simulation, when the run phase is complete.

ELABORATION AND SIMULATION PHASES

Now we can summarize the standard elaboration and simulation phases. The following callback hooks are called in the order shown for classes derived from ovms_component:

1. build Create components using new or the factory
2. connect Make port-to-export connections
3. end_of_elaboration After all connections have been hardened
4. start_of_simulation Pre-processing
5. run Simulation
6. extract Post-processing 1
7. check Post-processing 2
8. report Post-processing 3

THE TEST CLASS

We have looked at the verification environment, which generates a series of random transactions and drives them into the DUT, and at hooking up the verification environment to the DUT. We also emphasised in the previous article that a real verification environment would have to deal with checking and coverage collection across multiple, reusable verification components. Now we look at the test, which configures the verification environment to apply a specific stimulus to the DUT. A practical verification environment must allow you to choose between multiple tests, which must be selectable and runnable with minimal overhead.

The test is represented by a class derived from the methodology base class ovms_test, and is described line-by-line below:

```cpp
class my_test extends ovms_test;

'ovms_component_utils(my_test)

function new ( string name = "my_test", ovms_component parent = null );
  super.new ( name, parent );
endfunction: new

definition: my_env

virtual function void run;
  ovms_report_info ( "", "Called my_env::run" );
endtask: run

virtual function void report;
  ovms_report_info ( "", "Called my_env::report" );
endfunction: report

derived_class: my_env

endclass: my_env
```

The ovms_test class does not actually provide any functionality over-and-above an ovms_component, but the idea is that you use it as the base class for all user-defined tests.

```cpp
my_env m_env;

virtual function void build;
  super.build();
  $cast(m_env, create_component ( "my_env", "m_env" ) );
  set_global_timeout ( 1us );
endfunction: build

virtual function void connect;
  m_env.m_driver.m_dut_if = dut_if1;
endfunction: connect
```

The test instantiates the verification environment as a local component using the factory. It is reasonable that the test should depend on the environment, since each test will usually configure the environment to its own specific needs. The factory is used for consistency, although the flexibility provided by the factory is not needed in this particular case: we could equally well have used new to instantiate the environment.

The method set_global_timeout sets the watchdog timer referred to in the previous article to ensure that every run method will eventually return, particularly those run methods that are waiting on events or empty queues.
The connect method hooks up the DUT interface to the virtual interface in the driver, as discussed previously. This one assignment makes the connection between the structural SystemVerilog code, that is, modules and interfaces, and the class-based code of the verification environment.

```verilog
module top;
  ...;
  dut_if dut_if1();
  dut dut1 (.i_f(dut_if1));
class tiny_transaction extends my_transaction;
  ...;
endclass: tiny_transaction
class my_test extends ovm_test;
  ...
endclass: my_test
initial
  run_test();
endmodule: top
```

The simulator can now be started from the command line as follows:

```
QuestaSim> vsim top +OVM_TESTNAME=my_test
```

When the simulation runs, you should see the following:

- A message saying that my_test is being run
- Many of the callback hooks print out a message saying that they have been called
- The driver prints out a series of 10 messages showing that it is driving transactions into the DUT. Each transaction obeys the constraints set in tiny_transaction.
- The DUT prints out a series of 10 messages showing that it has received the corresponding commands

**THE CONFIGURATION INTERFACE**

We will now make some incremental improvements to the example to take it a little closer to being a reusable verification environment. In particular we will discuss the configuration interface.

The driver component in our example contains an infinite loop. The first improvement we will make is to limit the number of iterations by setting a parameter. Doing so will require us to introduce the OVM configuration interface, which is a very flexible and powerful mechanism for customizing the behaviour of a verification component. Here is the relevant part of the driver:

```verilog
module top;
  ...
  dut_if dut_if1();
  dut dut1 (.i_f(dut_if1));
class tiny_transaction extends my_transaction;
  ...
endclass: tiny_transaction
class my_test extends ovm_test;
  ...
endclass: my_test
initial
  run_test();
endmodule: top
```

This is a neat trick, because it means that the constraints on the address are part of the test, not hard wired into the verification environment. If generate_stimulus were to be called with a null argument instead of a transaction, only the constraints built into class my_transaction would be applied.

**RUNNING THE TEST**

We have got the verification environment connected to the DUT, and we have got a test, which could be one of many. The test is actually started by calling the run_test method. The name of the test to be run may be passed as an argument to run_test or may be given on the command line. Here is the outline of the top-level module:

```verilog
task run;
  tiny_transaction tx = new;
  m_env.m_stimulus.generate_stimulus ( tx, 10 );
  endtask: run
endclass: my_test
```

The run method kicks off the verification environment by calling the generate_stimulus method of class ovm_random_stimulus to generate exactly 10 transactions. The first argument tx will be used as the template for each random transaction as it is generated, and is used to pass test-specific constraints into the generator. The class tiny_transaction is specific to this test, and is defined as follows:

```verilog
class tiny_transaction extends my_transaction;
  `ovm_object_utils(tiny_transaction)
  constraint c_addr { addr >= 0; addr < 16; }
  constraint c_data { data >= 0; data < 16; }
endclass: tiny_transaction
```

This is a neat trick, because it means that the constraints on the address are part of the test, not hard wired into the verification environment. If generate_stimulus were to be called with a null argument instead of a transaction, only the constraints built into class my_transaction would be applied.
class my_driver extends ovm_driver;
    ...
    virtual task run;
    int count;
    if ( !get_config_int ( "iterations", count ) )
        // In the absence of a config setting, use a default value:
        count = 20;
    repeat ( count )
        begin
            ...
        end
    endtask: run
endclass: my_driver

The configuration interface comprises three set methods and three get methods that give access to a table of configuration settings. The method `get_config_int` searches the table for a field named “iterations” that is applicable to the current component instance. If a field of that name is found, the method sets the value of the second argument count to the value of the “iterations” field and returns true, otherwise it returns false. In this case, if the configuration field is not found, the driver gracefully reverts to using a default value of 20.

Entries are created in the table of configuration settings by calling one of the three methods `set_config_int`, `set_config_string` or `set_config_object`. Each entry consists of an instance name, a field name and a value. Values set by `set_config_int` can be retrieved by `get_config_int`, and the same for the other field types. Moreover, if there exists a field with a matching name, as created by the field automation macro `ovm_field_int`, then that field is also set to the same value. The existence of fields as created by `ovm_field_int` is not necessary for the operation of the `set_config_` / `get_config_` methods, but such fields are set as a side-effect of executing `set_config_`.

That is not the end of the story. The instance and field names in the configuration table can each include wildcard characters, “*” matching any sequence of characters and “?” matching any single character. These wildcard matches are greedy, meaning for example that the instance name “*” will match any instance name whatsoever, not just names restricted to a single hierarchical level. These wildcard matches have to be used with care, but they are very powerful.

The table of configuration settings is created before-the-fact, irrespective of the actual instance names and field names used in the component hierarchy. It contains raw text strings. As the verification component hierarchy gets elaborated, fields will be set if and only if their names match the strings in the configuration table. When combined with wildcards, this mechanism gives enormous flexibility in configuring the verification environment.

In our example, the “iterations” field is set by the test class as follows:

```verilog
class my_test extends ovm_test;
    ...
    my_env m_env;
    virtual function void build;
    ...
    set_config_int ( "m_env.\*", "iterations", 10 );
endclass: my_test
```

The instance name “m_env.\*” means that the setting will match any call to `get_config_int ( "iterations" ... )` within all component instances below the instance `m_env` in the component hierarchy.

We have seen an application of `set_config_` to set an iteration count for the driver. In general, `set_config_` may be used to configure verification components to meet the needs of the current verification environment, or to configure the verification environment to meet the needs of the current test. Configuration settings can be propagated down the verification component hierarchy using `get_config_` / `set_config_` or by means of wildcards, and can be tested conditionally within each verification component.

CONCLUSIONS

In this article we have examined the OVM elaboration and simulation phases, seen how to use a test class to separate test-specific information from the verification environment, and seen how the factory and configuration interface can be used to customize verification components according to their context. Tests, configurations, and the factory are key to enabling verification component re-use in OVM because they enable existing verification code to be re-purposed without any need to modify the original source code.

In the next article we will explore OVM sequences, which carry the idea of verification re-use over into the domain of sequential stimulus generation.
Closed Loop Requirement Verification
by Anupam Bakshi, Agnisys Technology

ABSTRACT:

The paper talks about a new unifying methodology that ensures closure to Verification planning and execution. Using this methodology and enabling tools, design/verification engineers and managers have a complete handle on the verification process. In a typical flow, a verification plan is created in Excel, Word, XML etc. and imported into the Simulation tool, as the verification process gets underway, the synchronization to the original plan is at best done manually. Further, you need help with understanding if the original requirements are being verified at all. This new methodology/tool, takes away the guessing game. In one document you describe the Verification Plan, check the latest Verification Status, know when you are done and what you are done with. This methodology works with Mentor’s UCDB and advanced verification technologies.

THE PROBLEM:

With regard to verification, we have heard the question before “Are we done yet?” Design/Verification Engineers need to know, Managers want to know and executives often ask where all the Verification effort is going and when the work will be completed. There are several tools/methodologies in the industry that can tell us with certainty how much of the code has been covered, how much of the functionality has been exercised.

However, shouldn’t the question really be, “Are we done with verifying the original requirements that we had established?” In other words, how close are we to achieving the goals we had setup initially to build the system according to the specification?

As the chip complexity grows, the need to make sure teams and resources are working towards the original requirements becomes critical. Requirement traceability is not just something that must be achieved in order to get the DO-254 certification; it is becoming a must-have methodology in order to get to market fast while using optimal resources. It shouldn’t be considered a burden, but a means to focus the team on the real objectives and get a clear understanding of where we are compared to our design verification goals.

Often times, the managers and executives on the team need to know what the status of the verification work is. Where are we in terms of the Verification plan that was put together months ago? Are the requirements being verified, are we getting any closer to achieving our objectives, or is it a case of diminishing returns. This need for transparency in the verification process is crucial for managing large scale verification projects.

More often than not teams report on their local achievements and the big-picture is lost in myriad of statistics, fancy reports and meetings.

Figure 1: Manual Verification Flow

OUR SOLUTION:

In our work as verification consultants, we needed a central place where all requirements, the verification plan and all associated verification progress was kept in an easy-to-access synchronized state, causing the least burden on the actual verification process, the verification environment and on the engineers. We needed a methodology that is not seen by the design and verification teams to be a chore, rather something that efficiently drives the process forward and helps in achieving our design verification objectives.

We have come up with a new methodology that revolves around a central active design document that was put together by the Design Team. This active document is used to capture all the requirements that the design team and the architects set out to implement. The Verification team refers to this active document quoting specific requirements and creates a Verification Plan. The beauty of this process is that as the Design Document evolves so does the Verification Plan. The two constantly remain in sync.

THE VERIFICATION PLAN CONTAINS THE FOLLOWING:

• pointers to the design document detailing the requirements
the actual coverage points as per the requirements
- details of various presumed or implemented assertions
- test benches
- details of test

As the requirements get modified, the Verification Plan automatically gets the updates. Verification engineers can quickly note the changes made to the plan and evaluate effectiveness of the plan and make suitable changes. Since the latest up-to-date requirements are right there in the same document, it gives a context and an anchor point for all design and verification activities.

Figure 2: Closed Loop Requirement Verification Methodology

We further instrumented this active document such that the results from the verification runs were tallied within the same document itself. The verification plan is then the focal point for the design and verification teams, as well as the management and executives. Everyone is able to relate to this document. Designers see their requirements that they have designed to, verification engineers can see their progress in verification space; and management/executives can see the overall progress, all in the same document. This is highly beneficial as just a look at the document clearly reveals where the holes in the verification plan are. Further, a closer examination also shows how to spend the least cycles in Verification for getting the desired features verified.

The active document as shown in figure 3, has requirements imported from the design document, the verification plan has been created in the same document. Further code coverage, functional coverage and assertion results are annotated onto the same document, in a clear and concise way.

Summary results clearly indicate what the status of the requirement verification effort is. It shows the useful information without cluttering it with gobs and gobs of numbers. This gives a clear idea about where the verification holes are and how much work still needs to be done. This leads to more informed schedules and resource allocation.

In principle, this methodology would work with any Vendor’s tools. However, Questa’s Verification Management facilities, especially the UCDB and its associated xml2ucdb, vcover and coverage utilities,

gave us all the hooks we needed to first send test data to the simulation environment and then get access to all the coverage data.

This methodology has many advantages, once all parties have agreed to use it. It was a central rallying point for the entire team, so any team member could see the effects of changes made to either the design or the verification plan. The team could collectively decide where more coverage or assertions were needed, and new members of the team found it advantageous to get up to speed quickly about their role in the verification process. Management and executives were able to open one single document and see the status without any ambiguity or delay.

Having the requirements in close proximity to the Verification plan helps us understand the relationship between the two. This automated flow, eliminates the need to do a manual comparison of the often changing design requirements to the verification plan. In addition the close proximity of the verification results to the requirements and the plan, give a warm and fuzzy about how well the original requirements have been verified with the current effort. Any short-comings can be promptly addressed.

This methodology brings the requirement out of the closet to the forefront of design verification, and helps close the loop on requirement verification.

About the Author: Anupam Bakshi is the Managing Director at Agnisys Technology Pvt. Ltd. Agnisys provides expert design/verification services and develops EDA tools. Anupam has spent close to two decades creating automation tools for Design/Verification process and managing large scale verification projects. He can be reached at ab@agnisys.us.
Concurrent Verification: Resolving Conflicting Goals
by Bruce Bergenfeld, Chief Technology Officer, Synterix Technology

INTRODUCTION
The purpose of this article is to point out a common issue that affects most ASIC development projects. Since Verification often consumes most of the man-hours of the ASIC development, typically double that of the RTL implementation, engineers and managers are trying to get as much mileage and value out of the verification as possible. If the verification environment can serve the needs of the designer in the early stages of RTL development, it is a big benefit.

GOALS AND CONFLICT
The goals and concerns of the verification team working towards tapeout signoff, and those of the RTL developer wishing to create robust design blocks, may sometimes diverge and even conflict.

VERIFICATION FOR TAPEOUT
Let’s define something that I would like to call, “Verification for Tapeout”. The primary goal of the verification process is to ensure that the design is correct before it is committed to silicon. This is Verification for Tapeout. Usually this is approached by using constrained random simulation and assertions combined with functional coverage. Using this methodology, if the functional coverage goals are met, and the checking is adequate, the verification for tapeout has been successful. These simulations try to cover all the important cases and combinations. It can take many weeks of massaging and adjusting the constraints to begin to reach the desired coverage. When coverage is not met, it is sometimes very difficult to pinpoint the root cause of the coverage hole.

VERIFICATION FOR RTL DEVELOPMENT
Often designers do not rely upon the HVL-based verification environment when they are in the early stages of development of their module. There are several reasons for this which include:

(1) Convenience – They are already developing the code in Verilog and they are already trying to compile the code to make sure it is clean of syntax errors. It is convenient to continue to use Verilog for the testbench and a set of simple tests.

(2) Directed-ness – The constrained random HVL-based verification environment is not directed enough to meet the designer’s very specific needs. The designer wants to make sure that all of the detailed functional code has been tested for at least a simple level of correctness.

(3) Too Complicated - The HVL-based verification environment is too complicated for the designer to deal with at this point. The designer wants to check his or her code to a certain level of correctness and move on to the next part of the design.

(4) Not Available - The HVL-based verification environment is not available yet.

I think that most designers can relate to these issues. The immediate goals of the RTL designer are just different than that of the tapeout.

GOALS IN CONFLICT: RTL VS. TAPEOUT
Designers that rely on Verilog for meeting their verification needs during development are paying a price for this. The approach of relying on a simple Verilog testbench for convenience is based upon a misconception to begin with. This so called “simple Verilog testbench” gradually but inevitably becomes a complicated and unwieldy testbench as it grows in functionality and complexity to try to test the RTL functions when they become available to test.

Those designers that prefer the simple Verilog testbench because of greater ability to direct the tests to test very specific functions are missing out on important reuse opportunities. After all of the work that goes into this Verilog testbench, it cannot be reused at the chip level or at any other level for that matter. Thirdly, it is likely that the Verilog testbench is not self-checking so it cannot be used for regression testing. For all of the work that goes into developing this Verilog testbench, it is in the end a one-shot, single-use endeavor.

If the motivation behind using Verilog is that the HVL-based verification environment is not meeting the needs of the designer because it is too complicated, then the HVL-based verification environment is missing an important opportunity to address the needs of the designer. This issue should be taken into account when the verification environment is in the planning stages.
If the motivation behind using Verilog is that the HVL-based verification environment is not yet available then there must be a problem of lack of resources or planning.

**DUT EXAMPLE – IEEE 802.3 10/100 MII MEDIA INDEPENDENT INTERFACE**

Let’s consider these conflicting goals in a bit more detail using the example of an MII interface for 10/100 Ethernet. The DUT is on the MAC (Media Access Controller) side of the MII such that packets are transferred from the PHY (Physical Media Device) to the MAC over the MII interface.

What would the goals of Verification for Tapeout be for such a design? The goal is to cover all combinations in every order that could possibly affect the functionality. This is done by randomizing everything that can be randomized. Meanwhile the checkers make sure that the DUT behaves as expected and intended.

The designer naturally agrees with this goal but has more pressing and more immediate needs. The designer ponders, “Does the feature I just added to my block really work?” To this end, the designer wants simple directed tests that check one feature at a time and then to add tests with increasing complexity. This gives the designer the confidence of a baseline of functionality. The designer wants to know what works and what does not work? When a test fails the designer thinks, “Where do I look to debug it?” These are the things that concern a designer at design time.

Considering such a DUT example, the Verification for Tapeout goal leads to a small number of very random tests that will exercise all combinations of inter-packet-gap, frame size, receive errors, CRC errors and various header fields that can affect the DUT behavior. Over time, a lot of simulation time, the functional coverage goals will be met and scenarios that have been considered or not considered will be covered.

Considering the DUT example above, what does the designer want? The designer would want something like the following list of directed tests:

- Test 1: Inject a single minimum size packet
- Test 2: Inject many minimum size packets with minimum IPG
- Test 3: Inject minimum size packets with IPG decreasing from 30 clocks down to 1 clock.
- Test 4: Inject a single maximum size packet
- Test 5: Inject many maximum size packets with IPG decreasing from 30 clocks down to 1 clock.

- Test 6: Inject many maximum size packets with IPG. Ping-pong between illegal and legal.
- Test 7: Inject many packets with size decreasing from legal minimum size down to zero byte payload.
- Test 8: Inject many packets with size. Ping-pong between legal minimum size and illegal runt frames.

These very directed tests help the designer to debug his code. In the earliest stages of RTL design, there is a lot of debug going on. Such tests are not very useful for tapeout because there would be a large number of them and each would be contributing very little to functional coverage. However, they would be useful for regression testing whenever the design or the verification environment changes.

**PROPOSED SOLUTION**

In order to address the needs of verification for RTL development and verification for tapeout, the HVL-based verification environment must be designed to meet the needs of both. This way the verification that is done for RTL development can be reused, can be included in regressions and can facilitate debugging when failures occur.

A great approach to address this problem is to utilize SystemVerilog and OVM. OVM provides a methodology to quickly ramp up a verification environment that can suit both the needs of the tape-out and the needs of the RTL design engineer process.

**OVM SEQUENCES SAVE THE DAY**

By utilizing OVM sequences, it becomes easy to write tests that are completely randomized and also to write tests that are very directed. In the solution proposed here a basic packet injection sequence was created that is called `rxmii_seq`.

```verilog
class rxmii_seq extends ovm_sequence;

function new(string name="rxmii_seq");
 super.new(name);
 endfunction

`ovm_sequence_utils(rxmii_seq, mii_sequencer)

mii_packet this_packet;
rand int unsigned ipg_del = 0;
constraint transmit_del_ct { (ipg_del inside {24:100}); };

rxmii_seq new("rxmii_seq");
mii_sequencer send(this_packet);

```
Parent sequences are created above this basic sequence in order to provide a way to tightly control the sequence constraints using the `ovm_do_with sequence invocation macro. The sequence below implements the directed test requirement of the designer, described as Test7. This parent sequence is called rxmii_designer7_seq.

The code examples shown were tested in an OVM–based verification environment using OVM version 1.1 and Questa version 6.4.

CONCLUSIONS AND RECOMMENDATIONS

OVM and OVM sequences are a powerful new tool for productive and effective verification. The sequences provide the flexibility in the verification environment to address the needs of the designer and the requirements for tapeout, while maintaining the proper separation between the verification environment and the verification tests by building the parent sequences above the basic sequences in a separate sequence library file.
Automated Register Abstraction Coming to an OVM Near You
by Jeremy Ralph, President, Productivity Design Tools Inc.

Picture this, you’re heading a specialized verification team, and you’ve just been handed a new chip with 57 different IP blocks, 5,871 addressable registers, and 22,273 bit-fields. Oh great, you think, what is the probability of the documentation matching the design? And once you get it in sync, how do you keep it that way?

Not to worry, there is a solution: abstracting the registers in a manner that maintains their synchronization with the corresponding RTL code, documentation, firmware, and more. Register-abstraction tools, such as SpectaReg™ SystemVerilog Register Abstraction (SVRA™) from PDTi, auto-generate SystemVerilog modules to provide everything needed for working with registers from a TLM verification standpoint. Since they are auto-generated, you can be confident that the abstracted registers match the single source spec and derived RTL — when the spec changes, so does everything that depends on it.

Addressable registers are important because they allow critical design decisions to be deferred until firmware development (compile-time or even runtime). Consequently, chips can be taped out sooner with more software programmability, making them more applicable to a wider range of applications. Registers, while obviously an on-chip interface, also provide an interface between the different developers; including RTL, software, verification, documentation, and validation engineers. It’s important to keep everyone on the same page. Errors and mis-synchronizations usually result in work-months of wasted effort repeating simulations, debugging, and respins.

This article explores how SpectaReg SVRA can be integrated with the Open Verification Methodology (OVM) to simplify register abstraction and reuse within an OVM verification environment. SpectaReg is a web application that manages addressable-register development and is deployable onsite and online. SpectaReg utilizes the network to easily scale and formalize addressable-register workflows and coding styles across teams and locations. Designs are collaboratively developed through a rich user interface and register-related deliverables are auto-generated as shown by Figure 1. SVRA was developed before the OVM initiative was launched and open sourced. So there is a non-OVM SVRA and a new SVRA for OVM in the works.

Figure 2 shows an auto-generated unit test environment constructed from reusable SVRA modules. This tests the generated RTL register interface at the lowest level of granularity. The same SVRA modules are reused for more complex hierarchies, as we will show.

**SVRA UNIT TESTING ENVIRONMENT**

Figure 2 shows an auto-generated unit test environment constructed from reusable SVRA modules. This tests the generated RTL register interface at the lowest level of granularity. The same SVRA modules are reused for more complex hierarchies, as we will show.
For both the random and directed tests (shown in Figure 2), Questa® provides useful and intuitive coverage graphs, as shown in Figure 3. The associated coverage points are automatically generated by SpectaReg as part of the SVRA register model.

Since SpectaReg generates the RTL, it knows the exact HDL path for each bit-field in the design. With this information, SVRA can provide backdoor access to the registers in zero time, so the simulation isn’t slowed down by the bureaucracy of handshaking all transactions through the bus interface. The goal for SVRA is to integrate with any backdoor register-access standard that comes out of the OVM standard.

**SVRA PER-COMPONENT REGISTER MODEL**

One of the more challenging aspects of verification modeling is keeping it synchronized with the underlying DUT. Verification typically begins before the design is complete, so you’re tracking a moving target to model or predict the expected behavior of the DUT as it evolves over time. A lack of synchronization results in added debugging and schedule slippage. Auto-generating SVRA and RTL simultaneously from a single source eliminates these synchronization headaches.

Central to the SpectaReg auto-generated SVRA is an object-oriented (OO) model of each component’s individually typed registers and bit-fields. The OO models include a variety of useful functions, coverage points, and random constraints. A separate functional coverage model is generated for tracking cross-coverage goals between inter-dependent bit-fields.
Why is OO modeling of registers important? Well, registers contain a lot of complexity that OO programming can abstract. A quote from Grady Booch (co-founder of Rational Software, IBM Fellow, and widely respected authority on software methodology) puts this into perspective:

… objects for me provide a means for dealing with complexity. The issue goes back to Aristotle: do you view the world as processes or as objects? Prior to the OO movement, programming focused on processes — structured methods for example. However, that system reached a point of complexity beyond which it could not go. With objects, we can build bigger, more complex systems by raising the level of abstraction — for me the real win of the object-oriented programming movement. 1

Reading and writing registers and accessing bit-fields and their encodings are supported through the OO model. The class structure aligns with the SPIRIT Consortium’s IP-XACT Component schema. The expected value of each DUT register is maintained by the model.

With OVM, the SVRA register model has the OVM object (ovm_object) as the root ancestor of its inheritance hierarchy. Transactions can be easily initiated using read and write methods provided by the register class. This utilizes OVM’s TLM classes (such as, ovm_transaction and ovm_tlm_fifo). OVM sequences and scenarios offer a way to reuse common patterns of register transactions across different test cases.

SVRA REGISTER SMART BUS MONITOR

For reverse transaction lookup, you need to figure out the target register, its bit-fields, and bit-field types and encodings. To provide such observability in a high-level way, the register smart monitor is auto-generated from SpectaReg. Given a transaction, the bus monitor looks up the target register from the register model. With OVM, the register smart bus monitor extends the OVM monitor (ovm_monitor).

SVRA REGISTER IO MONITOR

Each RTL register interface block has a unique register IO interface consisting of status inputs, control register outputs, handshaking, and memory-like ports and signals. This interface is shown between the register interface RTL and the application specific custom RTL in Figure 4.

To model the behavior of any component and its register values at any point in time, you need to predict the values of the status registers. If a test reads an unexpected status register, it gets flagged as an error. As the design evolves, pipelining stages often change, resulting in regression failure from temporal mismatches. To eliminate temporal mismatches, the register IO monitor provides a call back mechanism to alert the register model of any register IO changes. With OVM integration, the SVRA register IO monitor extends the OVM monitor class (ovm_monitor).

SVRA TRANSACTION AND DRIVER

The transactor and driver pattern converts between per-register object transactions and wire-level bus waveforms in a way that supports abstraction and cross-hierarchy reuse. This decoupling allows the underlying bus to be changed without affecting higher-level functional test cases. With SVRA-OVM integration, the SVRA driver extends the OVM driver class (ovm_driver).

REUSING THE SVRA ON MORE COMPLEX DUTS

The transactor and auto-generated SVRA model and monitors can be used at different levels of integration, as we will show.

COMPONENT FUNCTIONAL TESTING ENVIRONMENT USING SVRA

Figure 5 shows component functional testing using the SVRA and a functional model to check the component’s response to stimulus. Through the SVRA, custom tests read and write registers to verify the different configurations and states of the component. Useful functional coverage metrics are provided by auto-generated coverage points on each bit-field and cross coverage on interdependent bit-fields.
**SYSTEM INTEGRATION REGISTER AND FUNCTIONAL TESTING USING SVRA**

Figure 6 shows how the bus infrastructure for a system of N components can be verified using registers, by reusing the SVRA register tests. For higher-level functional testing, the SVRA portion of the environment is as shown in Figure 6, but with added custom functional tests and modeling. Higher-level tests utilize and extend SVRA for all register aspects.

**CONCLUSION**

This article has provided an overview of how SpectaReg works to auto-generate synchronized register deliverables. As the design changes over time, synchronization of the different deliverables (such as RTL, SVRA, firmware, and documentation) is maintained. For verification engineers, SVRA abstracts registers to simplify their inherent complexities while providing verification modules that can be used and reused at different levels of hierarchical testing. SVRA integration with OVM provides an advanced verification environment in the context of an open and growing methodology and ecosystem.

To arrange a SpectaReg demo and evaluation, contact PDTi at http://productive-eda.com/contact-PDTi.

**NOTE**

I should like to share some of our team’s recent good experiences with the newly released MVC Verification components library. We have found the MVCs to be extremely useful in solving customer verification problems.

The primary reason for using any verification IP is that it should save you the time and effort to develop and verify a model for your verification environment. At the very least, it should come with a means of generating some stimulus and a means of being able to check a protocol. We were not disappointed with the MVC Verification IP library since each VIP comes as a comprehensive kit of transactors, monitors, functional coverage monitors, responders, masters, slaves, stimulus classes and a UCDB ready functional coverage plan. There are also examples that illustrate how to use the MVC in different situations and a number of the example classes provided can quickly be adapted to solve specific project verification problems. We have used MVCs in customer verification environments based on both the AVM and the OVM and well as in traditional VHDL and Verilog verification environments.

Our first experience of using one of the MVCs was when we were contacted by a customer who wanted to ensure that their AXI bus inter-connect was fully compliant to the AXI protocol. They had run many tests and had used other third party VIP but were not convinced that they had seen all possible types of AXI cycle. We downloaded the MVC library and went on site to see what we could do. We were able to integrate the MVC into their VHDL verification environment in a short space of time and then set about generating random stimulus to validate their inter-connect. We adapted one of the examples supplied as part of the kit to generate random stimulus according to the memory map of the interconnect (i.e. This range of memory does not support this type of access, but this range does). We set up functional coverage monitoring to check for all types of AXI cycle and linked this to the UCDB ready functional coverage plan that was supplied with the MVC. Finally, we developed a scoreboard to check for correct operation. We were very quickly able to run test cases that generated all possible AXI cycles and started to find problems with the inter-connect that had not previously been detected. One of these was a third order address translation issue that only occurred between one port of the inter-connect and another just after a burst had completed.

In this first example, the MVC was used as a BFM in a pre-existing VHDL test bench. The MVC was encapsulated within an AVM environment and this in turn was encapsulated within a verilog module that was then instantiated as an AXI master in the VHDL test bench. Using Questa, this was very straight-forward to do and we were able to generate different test cases simply by changing constraints for the MVC stimulus generation. Prior to this short burst of activity, the customer had been sceptical of constrained random verification, but this experience convinced them of its value.
We then went on to work with another project team from the same customer to validate an OCP inter-connect. The project requirements were similar, but this time we had the chance to create an OVM test bench that went around their inter-connect. In addition to generating stimulus with the MVC, we used the QVL OCP monitor to check the bus protocol. Again, we adapted the functional coverage plan supplied with the MVC to prove that we had covered all the required protocol cases. Using the OCP MVC example code we were able to completely verify the customers interconnect in a few weeks, finding a number of show-stopper bugs along the way. These included an address mapping issue which caused a posted write to an address sent from one OCP master to arrive at the wrong output of the switch in the cycle after a non-posted write to another region of the memory map had completed.

Our next MVC adventure came when we were approached by a customer to develop a way of being able to drive MVCs using a software interface in an OVM environment. Their requirement was to be able to test the software interface to their hardware at an early stage, using a MVC to drive transactions on the target on-chip bus based on a software API that could be compiled with their test software. The test programs compiled against the API needed to run on a host machine in parallel with the simulation. At the same time they wanted to be able to generate directed or constrained random stimulus using SystemVerilog with the same OVM environment, using either software or SystemVerilog to generate stimulus, or some combination of the two.

We developed a generic API that could support most kinds of bus transaction - reads, writes, bursts of reads and writes etc – with the knowledge that the API could be used to target different on-chip buses by substituting the appropriate MVC for the bus in the design. The DPI was used to implement the bridge between the API and the SystemVerilog simulation. However, since the DPI can only make function calls from c to SystemVerilog or from SystemVerilog to c, we had to spawn a server program from the simulation that would listen for remote process calls from the test software and then translate them into DPI calls to send request transactions. The server program was also responsible for listening out for response transactions coming from the SystemVerilog side of the DPI.

The SystemVerilog side of the DPI interface was implemented in a module that communicated with the rest of the OVM verification environment using a pair of TLM FIFOs. When an API call requested a transaction on the bus, a data structure would be transferred across the DPI and then converted to a transaction that was put into a tlm_fifo contained within an extended scenario controller class. A server scenario ran on the scenario channel which had the responsibility of checking for API induced transactions and then passing them on
to the scenario driver through the scenario controller. Any response transactions from the MVC would then be sent back to the DPI server module by the server scenario and in turn these would be sent back to the software that had instigated the API call.

The transactional interface to MVCs consists of several ports that take and emit transactions of different types ranging from transactions requesting phase level activity to a complete burst transfer transaction. Each MVC has its own distinct set of transactions which are aligned with the characteristics of the interface so in our solution the role of the scenario driver is to do the translation between the generic transactions and the MVC specific transactions.

A coding trick we used to simplify the design of the scenario TLM communication chain was to parameterise the tlm_fifos, the scenario, scenario_controller and the scenario_driver with an ovm_transaction type rather than a specific transaction type. This allowed us to use polymorphism to send a number of different transaction types over the same TLM path to accommodate request and response traffic for different types of phase, transaction and burst transaction types. Both ends of the communication chain were aware of the types of transaction that could be used and by testing the type property of the received transaction they could cast it to the required transaction type and then deal with it appropriately.

In order to accommodate the use of the MVC with SystemVerilog stimulus we developed a Scenario base class that runs on the same scenario_controller and sends transactions to the MVC. Extensions of this base class can then be used in parallel with the server scenario that looks after TLM traffic from the software API. Arbitration between the two stimulus sources is handled by the scenario_controller and the arbitration can be set up to give priority to one kind of access over another.

In summary, our experience of using the MVCs has been very positive and they have proved to be a useful tool in providing verification solutions.